O'REILLY



Early Release
RAW & UNEDITED



Data Visualization with Python & JavaScript

SCRAPE, CLEAN & TRANSFORM YOUR DATA



Kyran Dale

data visualization with python and javascript

Crafting a Data-visualisation Toolchain for the Web

Kyran Dale

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Data Visualization with Python and JavaScript

by Kyran Dale

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editors: Dawn Schanafelt and Meghan Blanchette
Production Editor: FILL IN PRODUCTION EDITOR
Copyeditor: FILL IN COPYEDITOR
Proofreader: FILL IN PROOFREADER
Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

January -4712: First Edition

Revision History for the First Edition

2016-02-22: First Early Release
2016-03-21: Second Early Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491956434* for release details.

978-1-491-95643-4

[FILL IN]

Table of Contents

Introduction

This book aims to get you up to speed with what is, in my opinion, the most powerful data-visualisation stack going: Python and JavaScript. You'll learn enough of big libraries like Pandas and D3 to start crafting your own web data-visualisations and refining your own toolchain. Expertise will come with practice but this book presents a shallow learning curve to basic competence.

> If you're reading this in Early Release form I'd love to hear any feedback you have. Please post it to *pyjsdataviz@kyrandale.com*. Thanks a lot, Kyran.
>
> You'll also find a working copy of the Nobel-visualisation the book literally and figuratively builds towards at *http://kyrandale.com/static/pyjsdataviz/index.html*.

The bulk of this book tells one of the innumerable tales of data-visualisation, one carefully selected to showcase some powerful Python and JavaScript libraries or tools which together form a toolchain. This toolchain gathers raw, unrefined data at its start and delivers a rich, engaging web-visualisation at its end. Like all tales of data-visualisation it is a tale of transformation, in this case transforming a basic Wikipedia list of Nobel prize-winners into an interactive visualisation, bringing the data to life and making exploration of the prize's history easy and fun.

A primary motivation for writing the book is the belief that, whatever data you have, whatever story you want to tell with it, the natural home for the visualizations you transform it into is the web. As a

delivery platform it is orders of magnitude more powerful than what came before and this book aims to smooth the passage from desktop or server-based data analysis and processing to getting the fruits of that labour out on the web.

But the most ambitious aim of this book is to persuade you that working with these two powerful languages towards the goal of delivering powerful web-visualisations is actually fun and engaging.

I think many potential data-viz programmers assume there is a big divide, called *Web Development*, between doing what they would like to do, which is program in Python and JavaScript. Web-dev involves loads of arcane knowledge about markup-languages, style-scripts, administration etc. and can't be done without tools with strange names like *Gulp* or *Yeoman*. I aim to show that these days that big divide can be collapsed to a thin and very permeable membrane, allowing you to focus on what you do well, programming stuff (see Figure P-1) with minimal effort, relegating the web-servers to data-delivery.

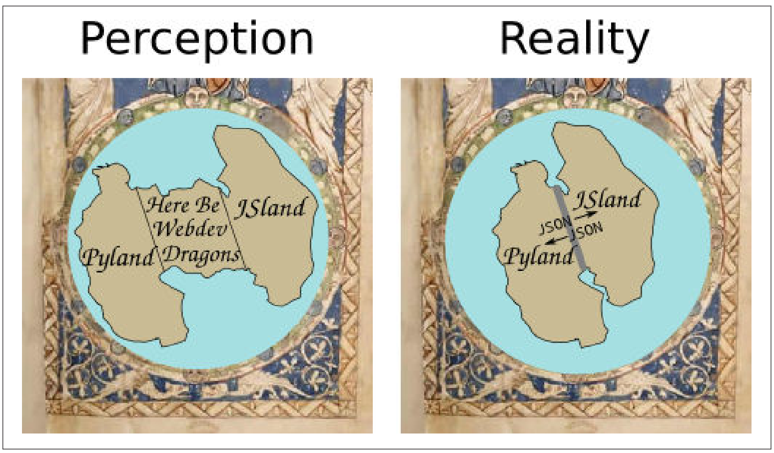


Figure P-1. Here be web-dev dragons

Who This Book is For

First off, this book is for anyone with a reasonable grasp of Python or JavaScript who wants to explore one of the most exciting areas in the data-processing ecosystem right now, the exploding field of data-visualisation for the web. It's also about addressing some specific pain-points which in my experience are quite common.

When you get commissioned to write a technical book, chances are your editor will sensibly caution you to think in terms of 'pain points' that your book aims to address. The two key pain points of this book are best illustrated by way of a couple of stories, one my own, the other one that has been told to me in various guises by JavaScripters I know.

Many years ago, as an academic researcher, I came across Python and fell in love. I had been writing some fairly complex simulations in C(++) and Python's simplicity and power was a breathe of fresh air from all the boilerplate, Makefiles, declarations and definitions and the like. Programming was fun, Python the perfect glue, playing nicely with my C(++) libraries (Python wasn't then and still isn't a speed demon) and doing, with consummate ease, all the stuff that in low level languages is such a pain, e.g. file I/O, database access, serialisation etc.. I started to write all my graphical user interfaces (GUIs) and visualisations in Python, using *wxPython*, *PyQt* and a whole load of other refreshingly easy toolsets. Now there's some stuff there that I think is pretty cool but I doubt I'll ever get around to the necessary packaging, version checking and various other hurdles to distribution, so no-one else will ever see it.

At the time there existed what in theory was the perfect universal distribution system for the software I'd so lovingly crafted, namely the web-browser. Available on pretty much every computer on earth, with its own built-in, interpreted programming language, write once, run everywhere. But everyone knew that *a.* Python doesn't play in the web-browser's sandpit and *b.* browsers were incapable of ambitious graphics and visualisations, being pretty much limited to static images and the odd jQuery transformation. JavaScript was a 'toy' language tied to a very slow interpreter good for little *DOM* tricks but certainly nothing approaching what I could do on the desktop with Python. So that route was discounted, out of hand. My visualisations wanted to be on the web but there was no route through.

Fast forward a decade or so and, thanks to an arms race initiated by Google and their V8 engine, JavaScript is now orders of magnitude faster, in fact it's now an awful lot faster than Python[1]. HTML has also tidied up its act a bit, in the guise of HTML5. It's a lot nicer to

1 See here for a fairly jaw-dropping comparison.

work with, with much less boilerplate. What were loosely followed and distinctly shaky protocols like Scalable Vector Graphics (SVG) have firmed up nicely thanks to powerful visualisation libraries, D3 being preeminent. Modern browsers are obliged to work nicely with SVG and, increasingly, 3D in the form of *WebGL* and its children such as *THREE.js*. Those visualisations I was doing in Python are now possible on your local web-browser and the payoff is that, with very little effort, they can be made accessible to every desktop, laptop, smartphone and tablet in the world.

So why aren't Pythonistas flocking to get their data out there in a form they dictate? After all, the alternative to crafting it yourself is leaving it to somebody else, something most data-scientists I know would find far from ideal. Well, first there's that term *Web Development*, connoting complicated markup, opaque stylesheets, a whole slew of new tools to learn, IDEs to master. And then there's JavaScript itself, a strange language, thought of as little more than a toy until recently and having something of the neither fish nor fowl to it. I aim to take those pain-points head-on and show that you can craft modern web-visualisations (often single page apps) with a very minimal amount of HTML and CSS boilerplate, allowing you to focus on the programming, and that JavaScript is an easy leap for the Pythonista, having a lot in common. But you don't have to leap, Chapter 2 is a language-bridge, which aims to help Pythonistas and JavaScripters bridge the divide between the languages by highlighting common elements and providing simple translations.

The second story is a common one I run into among JavaScript data-visualiers I know. Processing data in JavaScript is far from ideal. There are few heavyweight libraries and although recent functional enhancements to the language make data-munging much more pleasant, there's still no real data-processing ecosystem to speak of. So there's a distinct asymmetry between the hugely powerful visualisation libraries available, D3 as ever paramount, and the ability to clean and process any data delivered to the browser. All of this mandates doing your data-cleaning, processing and exploration in another language or with a toolkit like Tableau and this often devolves into piecemeal forays into vaguely remembered Matlab, the steepish learning curve that is R or a Java library or two.

Toolkit's like Tableau, although very impressive, are often, in my experience, ultimately frustrating for programmers. There's no way to replicate in a GUI the expressive power of a good, general pur-

pose programming language. Plus, what if you want to create a little web-server to deliver your processed data? That means learning at least one new web-dev capable language.

In other words, JavaScripters starting to stretch their data visualisation are looking for a complementary data-processing stack which requires the least investment of time and has the shallowest learning curve.

Minimal requirements to use the book

I always feel reluctant placing restrictions on people's explorations, particularly in the context of programming and the web, which is chock full of auto-didacts (how else would one learn, the halls of academe being lightyears behind the trend?), learning fast and furiously, gloriously uninhibited by the formal constraints that used to apply to learning. Python and JavaScript are pretty much as simple as it gets, programming language wise, and are both top candidates for best first language. There isn't a huge cognitive load in interpreting the code.

In that spirit, there are expert programmers who, without any experience of Python and JavaScript, could consume this book and be writing custom libraries within a week. These are also the people most likely to ignore anything I write here so good luck to you people if you decide to make the effort.

For beginner programmers, fresh to Python or JavaScript, this book is probably too advanced for you and I'd recommend taking advantage of the plethora of books, web-resources, screencasts and the like that make learning so easy these days. Focus on a personal itch, a problem you want to solve and learn to program by doing - it's the only way.

For people who have programmed a bit in either Python or JavaScript, my advised threshold to entry is that you have used a few libraries together, understand the basic idioms of your language and can look at a piece of novel code and generally get a hook on what's going on. i.e. Pythonistas who can use a few modules of the standard library and JavaScripters who can not only use Jquery but understand some of its source-code.

Why Python and JavaScript?

Why JavaScript is an easy question to answer. For now and the foreseeable future there is only one first class, browser-based programming language. There have been various attempts to extend, augment and usurp but good old plain vanilla JS is still preeminent. If you want to craft modern, dynamic, interactive visualisations and, at the touch of a button, deliver them to the world, at some point you are going to run into JavaScript. You might not need to be a zen master but basic competence is a fundamental price of entry into one of the most exciting areas of modern data science. This book hopes to get you into the ballpark.

Why not Python on the browser?

There are currently some very impressive initiatives aimed at enabling Python produced visualisations, often built on Matplotlib, to run in the browser. This is achieved by converting the Python code into JavaScript based on the `canvas` or `svg` drawing contexts. The most popular and mature of these are Bokeh and the recently open-sourced Plotly. IPython's Jupyter project. While these are both brilliant initiatives, I feel that in order to do web-based data-viz you have to bite the JavaScript bullet to exploit the increasing potential of the medium. That's why, along with space constraints, I'm not covering the Python to Javscript dataviz converters.

While there is some brilliant coding behind these JavaScript converters and many solid use-cases, they do have big limitations:

- Automated code-conversion may well do the job but the code produced is usually pretty impenetrable for a human being.
- Adapting and customising the resulting plots using the powerful browser-based JavaScript development environment is likely to be very painful.
- You are limited to the subset of plot types currently available in the libraries.
- Interactivity is very basic at the moment. Stitching this together is better done in JavaScript, using the browser's dev-tools.

Bear in mind that the people building these libraries have to be JavaScript experts so if you want to understand anything of what they're

doing and eventually express yourself then you'll have to get up to scratch with some JavaScript.

My basic take-home with Python-to-JavaScript conversion is that it has its place but would only be generally justified were JavaScript ten times harder to program than it is. The fiddly, iterative process of creating a modern browser-based data-visualisation is hard enough using a first-class language without having to negotiate an indirect journey through a second-class one.

Why Python for data-processing

Why you should choose Python for your data-processing needs is a little more involved. For a start there are good alternatives as far as data-processing is concerned. Let's deal with a few candidates for the job, starting with the enterprise behemoth Java:

Java

Among the other main, general-purpose programming languages, only *Java* offers anything like the rich ecosystem of libraries that Python does, with considerably more native speed too. But while *Java* is a lot easier to program in than, say, C++ it isn't, in my opinion, a particularly nice language to program in, having rather too much in the way of tedious boilerplate and excessive verbiage. This sort of thing starts to weigh after a while and makes for a hard slog at the code-face. As for speed, Python's default interpreter is slow but Python is a great *glue* language, which plays nicely with other languages. This ability is demonstrated by the big Python data-processing libraries like *Numpy* (and its dependent *Pandas*), *Scipy* and the like, which use `C(++)` and Fortran libraries to do the heavy lifting while providing the ease of use of a simple, scripting language.

R

The Venerable R has, until recently, been the tool of choice for many data-scientists and is probably Python's main competitor in the space. Like Python, R benefits from a very active community, some great tools like the plotting library ggplot and a syntax specially crafted for data-science and statistics. But this specialism is a double-edged sword. Because R was developed for a specific purpose, it means that if, for example, you wish to write a web-server to serve

your R processed data, you have to skip out to another language, with all the attendant learning overheads, or try and hack something together, round hole, square-peg wise. Python's general purpose nature and it's rich ecosystem mean one can do pretty much everything required of a data-processing pipeline (JS visuals aside) without having to leave its comfort zone. Personally, it is a small sacrifice to pay for a little syntactic clunkiness.

Others

There are other alternatives to doing your data-processing with Python but none of them come close to the flexibility and power afforded by a general purpose, easy to use programming language with a rich ecosystem of libraries. While, for example, mathematical programming environments such as Matlab and Mathematica have active communities and a plethora of great libraries, they hardly count as general purpose, designed to be used within a closed garden. They are also proprietary, which means a significant initial investment and a different vibe to the Python's resoundingly open-source environment.

GUI-driven data-viz tools like Tableau are great creations but will quickly frustrate someone used to the freedom to programming. They tend to work great as long as you are singing from their song-sheet, as it were. Deviations from the designated path get painful very quickly.

Python's getting better all the time

As things stand, I think a very good case can be made for Python being the budding data-scientist's language of choice. But things are not standing still, in fact Python's capabilities in this area are growing at an astonishing rate. To put it in perspective, I have been programming in Python for over fifteen years, have grown used to being surprised if I can't find a Python module to help solve a problem at hand, but I find myself surprised at the growth of Python's data-processsing abiliites, with a new, powerful library appearing weekly. To give an example, Python has traditionally been weak on statistical analysis libraries, with R being far ahead here. Recently a number of powerful modules, such as *StatsModel* have started to close this gap fast.

So Python is a thriving data-processing ecosystem with pretty much unmatched general purpose and it's getting better week on week. It's understandable why so many in the community are in a state of such excitement - it's pretty exhilarating.

As far as visualisation in the browser, the good news is that there's more to JavaScript than its privileged, nay exclusive place in the web-ecosystem. Thanks to an interpreter arms race which has seen performance increase in staggering leaps and bounds and some powerful visualisation libraries, such as D3, which would complement any language out there, JavaScript now has serious chops.

In short, Python and JavaScript are a wonderful complement for data-visualisation on the web, each needing the other to provide a vital missing component.

What You'll Learn

There are some big Python and JavaScript libraries in our dataviz toolchain and comprehensive coverage of them all would require a number of books. Nevertheless I think that the fundamentals of most libraries, and certainly the ones covered here, can be grasped fairly quickly. Expertise takes time and practice but the basic knowledge needed to be productive is, so to speak, low-hanging fruit.

In that sense this book aims to give you a solid backbone of practical knowledge, strong enough to take the weight of future development. I aim to make the learning curve as shallow as possible and get you over the initial climb, with the practical skills needed to start refining your art.

This book emphasises pragmatism and best-practice. It's going to cover a fair amount of ground and there isn't enough space for too many theoretical diversions. I will aim to cover those aspects of the libraries in the toolchain that are most commonly used, and point you to resources for the other stuff. Most libraries have a hard-core of functions, methods, classes and the like that are the chief, functional subset. With these at your disposal you can actually do stuff. Eventually you find an itch you can't scratch with those at which time good books, documentation and online forums are your friend.

The Choice of Libraries

I had three things in mind while choosing the libraries used in the book.

1. Open-source and free as in beer - you shouldn't have to invest any extra money to learn with this book.
2. Longevity - generally well-established, community-driven and popular.
3. Best-of-breed, allowing for *2.* - at the sweet-spot between popularity and utility.

The skills you learn here should be relevant for a long time. Generally the obvious candidates have been chosen, libraries that write their own ticket as it were. Where appropriate I will highlight the alternative choices and give a rationale for my selection.

Preliminaries

A few preliminary chapters are needed before beginning the transformative journey of our Nobel data-set through the toolchain. These aim to cover the basic skills necessary to make the rest toolchain chapters run more fluidly. The first few chapters cover the following:

- Building a Language bridge between Python and JavaScript.
- Covering the Basic Web-dev needed by the book.
- How to pass data around with Python, through various file-formats and databases.

These chapters are part-tutorial, part-reference and it's fine to skip straight to the beginning of the toolchain, dipping back where needed.

The Dataviz Toolchain

The main part of the book demonstrates the data-visualisation toolchain, which follows the journey of a data-set (of Nobel-prize winners) from raw, freshly scraped data to engaging, interactive JavaScript visualisation. During the process of collection, refinement and transformation a number of big libraries are demonstrated, summarised in Figure P-2. These libraries are the industrial lathes of

our toolchain, rich, mature tools which demonstrate the power of the Python + JavaScript dataviz stack. Here's a brief introduction to the five stages of our toolchain and their major libraries.

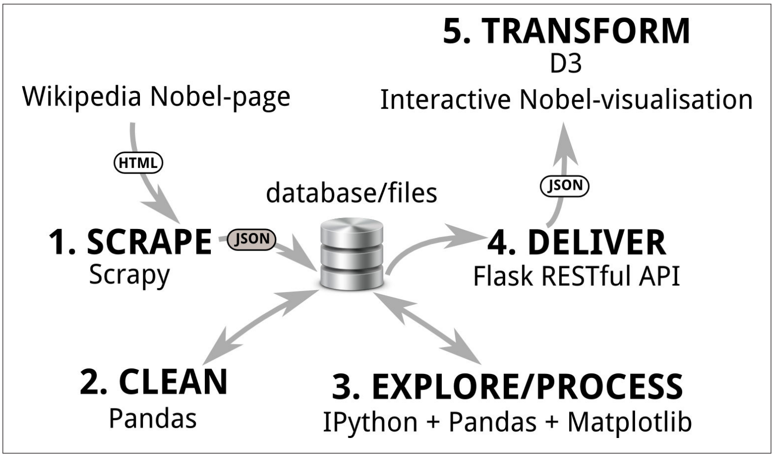


Figure P-2. The Data-viz Toolchain

1. Scraping data with Scrapy

The first challenge for any data-visualiser is getting hold of the data they need, whether by request or to scratch a personal itch. If you're very lucky this will be delivered to you in pristine form but more often than not you have to go find it. I'll cover the various ways you can use Python to get data off the web (e.g. web-APIs or google-spreadsheets) but the Nobel-prize data-set for the toolchain demonstration is scraped[2] from its Wikipedia pages using Scrapy.

Python's Scrapy is an industrial strength scraper that does all the data-throttling, media pipelining etc. that are indispensable if you plan on scraping significant amounts of data. Scraping is often the only way to get the data you are interested in and once you've mastered Scrapy's workflow all those previously off-limits datasets are only a spider[3] away.

2 Web scraping is a computer software technique to extract information from websites, usually involving getting and parsing web-pages.

3 Scrapy's controllers are called spiders.

2. Cleaning data with Pandas

The dirty secret of data-viz is that pretty much all data is dirty and turning it into something you can use may well occupy a lot more time than anticipated. This is an unglamourous process which can easily steal over half your time. Which is all the more reason to get good at it and use the right tools.

Pandas is a huge player in the Python data-processing ecosystem. It's a Python data-analysis library whose chief component is the DataFrame, essentially a programmatic spreadsheet. Pandas extends Numpy, Python's powerful numeric library, into the realm of heterogeneous data-sets, the kind of categorical, temporal, ordinal etc. information that data-visualisers have to deal with. As well as being great for interactively exploring your data (using its built-in Matplotlib plots), Pandas is well-suited to the drudge-work of cleaning data, making it easy to locate duplicate records, fix dodgy date-strings, find missing fields etc..

3. Exploring data with Pandas and Matplotlib

Before beginning the transformation of your data into a visualisation you need to understand it. The patterns, trends, anomalies etc. hidden in the data will inform the stories you are trying to tell with it, whether that's explaining a recent rise in year-on-year widget sales or demonstrating global climate change.

In conjunction with IPython, the Python interpreter on steroids, Pandas and Matplolib (with additions such as Seaborn) provide a great way to explore your data interactively, generating rich, inline plots from the command-line, slicing and dicing your data to reveal interesting patterns. The results of these explorations can then be easily saved to file or database to be passed on to your JavaScript visualisation.

4. Delivering your data with Flask

Once you've explored and refined your data you'll need to serve it to the web-browser, where a JavaScript library like D3 can transform it. One of the great strengths of using a general purpose language like Python is that it's as comfortable rolling a web-server in a few lines of code as it is crunching through large datasets with special-

purpose libraries like Numpy and Scipy[4]. Flask is Python's most popular lightweight server and is perfect for creating a small, RESTful[5] APIs which can be used by JavaScript to get data from the server, in files or databases, to the browser. As I'll demonstrate, you can roll a RESTful API in a few lines of code, capable of delivering data from SQL or NoSQL databases.

5. Transforming the data into interactive visualisations with D3

Once the data is cleaned and refined, we have the visualisation phase, where selected reflections of the data-set are presented, ideally allowing the user to explore them interactively. Depending on the data this might involve barcharts, maps or novel visualisations.

D3 is JavaScript's powerhouse visualisation library, arguably one of the most powerful visualisation tools irrespective of language. We'll use D3 to create a novel Nobel-prize visualisation with multiple elements and user interaction, allowing people to explore the dataset for items of interest. D3 can be challenging to learn but I hope to bring you quickly up to speed, ready to start honing your skills in the doing.

Smaller Libraries

As well as the big libraries covered, there is a large supporting cast of smaller libraries. These are the indispensable smaller tools, the hammers, spanners etc. of the toolchain. Python in particular has an incredibly rich ecosystem, with small, specialised libraries for almost every conceivable job. Among the strong supporting cast, some particularly deserving of mention are:

- `requests`: Python's go-to HTTP library, fully deserving its motto 'HTTP for humans'. requests is far superior to urllib2, one of Python's included batteries.

4 The Scientific Python library, part of the Numpy ecosystem.

5 REST is short for Representational State Transfer, the dominant style for HTTP-based web-APIs and much recommended.

- `SQLAlchemy`: the best Python SQL Toolkit and Object Relational Mapper (ORM) there is. It's feature rich and makes working with the various SQL-based databases a relative breeze.
- `Seaborn`: a great addition to Python's plotting powerhouse Matplotlib, adding some very useful plot-types including some statistical ones of particular use to data-visualisers. It also adds arguably superior aesthetics, over-riding the Matplotlib defaults.
- `crossfilter`: although JavaScript's data-processing libraries are a work-in-progress, a few really useful ones have emerged recently with `crossfilter` being a stand-out. It enables very fast filtering of row-columnar datasets and is ideally suited to data-viz work, unsurprising as one of it's creators is Mike Bostock, the father of D3.

A Little Bit of Context

This is a practical book and assumes that the reader has a pretty good idea what he or she wants to visualise, how that visualization should look and feel and a desire to get cracking on, unencumbered by too much theory. Nevertheless, drawing on the history of data-visualisation can both clarify the central themes of the book and add valuable context. It can also help explain why now is such an exciting time to be entering the field, as technological innovation is driving novel data-viz forms and people are grappling with the problem of presenting the increasing amount of multi-dimensional data generated by the internet.

Data visualisation has an impressive body of theory behind it and there are some great books out there that I would recommend you read (see ??? on page 21 for a little selection). The practical benefit of understanding the way humans visually harvest information cannot be overstated. It can be easily demonstrated, for example, that a pie-chart is almost always a bad way of presenting comparative data and a simple bar-chart far preferable. By conducting psychometric experiments, we now have a pretty good idea how to trick the human visual system and make the relationships in the data harder to grasp. Conversely we can show that some visual forms are close to optimal for amplifying contrast. The literature, at its very least, provides some useful rules of thumb which suggest good candidates for any particular data narrative.

In essence good data-viz tries to present data, collected from measurements in the world (empirical) or maybe the product of abstract mathematical explorations (e.g. the beautiful fractal patterns of the *Mandlebrot set*, in such a way as to draw out or emphasise any patterns or trends that might exist. These patterns can be simple, e.g. average weight by country, or the product of sophisticated statistical analysis, e.g. data-clustering in a higher dimensional space.

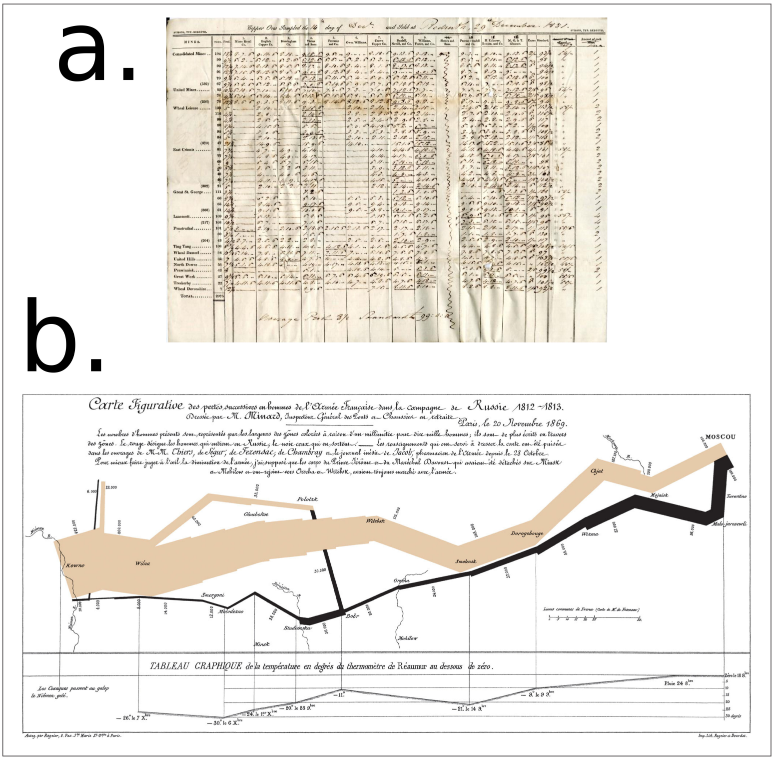


Figure P-3. (a) An early spreadsheet (b) Joseph Minard's visualisation of Napoleon's Russian campaign of 1812

In its untransformed state, we can imagine this data floating as a nebulous cloud of numbers or categories. Any patterns or correlations are entirely obscure. It's easy to forget but the humble spreadsheet (Figure P-3 a.) is a data-visualisation, the ordering of data into row-columnar form an attempt to tame it, make its manipulation easier and highlight discrepancies (e.g. actuarial book-keeping etc..). Of course, most people are not adept at spotting patterns in rows of numbers, so more accessible, visual forms were developed to engage

with our visual-cortex, the prime human conduit for information about the world. Enter the bar-chart, pie-chart[6], line-chart etc. More imaginative ways were employed to distil statistical data in a more accessible form, one of the most famous being Charles Joseph Minard's visualisation of Napoleon's disastrous Russian campaign of 1812 (Figure P-3 b.).

The tan colored stream in Figure P-3 b. shows the advance of Napoleon's army on Moscow, the black line the retreat. The thickness of the stream represents the size of Napoleaon's army, thinning as casualties mounted. A temperature chart below is used to indicate the temperature at locations along the way. Note the elegant way in which Minard has combined multi-dimensional data (casualty statistics, geographical location and temperature) to give an impression of the carnage which would be hard to grasp in any other way (imagine trying to jump from a chart of casualties to a list of locations and make the necessary connections). I would argue that the chief problem of modern interactive data-viz is exactly that faced by Minard: how to move beyond conventional one-dimensional bar-charts etc. (perfectly good for many things) and develop new way to communicate cross-dimensional patterns effectively.

Until quite recently, most of our experience of charts was not much different from those of Charles Minard's audience. They were pre-rendered, inert and showed one reflection of the data, hopefully an important and insightful one but nevertheless under total control of the author. In this sense the replacement of real ink-points with computer screen pixels was only a change in the scale of distribution.

The leap to the internet just replaced newsprint with pixels, the visualisation still being unclickable and static. Recently the combination of some powerful visualisation libraries (D3 preeminent among them) and a massive improvement in JavaScript's performance have opened the way to a new type of visualization, one that is easily accessible, dynamic and actually encourages exploration and discovery. The clear distinction between data exploration and presentation is blurred. This new type of data visualisation is the focus of this book and the reason why data-viz for the web is such an exciting area right now. People are trying to create new ways to visualize data

6 William Playfair's *Statistical Breviary* of 1801 having the dubious distinction of origin

and make it more accessible/useful to the end-user. This is nothing short of a revolution.

Summary

Dataviz on the web is an exciting place to be right now with innovations in interactive visualisations coming thick and fast, many (if not most) of them being developed with D3. JavaScript is the only browser-based language so the cool visuals are by necessity being coded in it (or converted into it). But JavaScript lacks the tools or environment necessary for the less dramatic but just as vital element or modern data-viz, the aggregation, curation and processing of the data. This is where Python rules the roost, providing a general-purpose, concise and eminently readable programming language with access to an increasing stable of first-class data-processing tools. Many of these tools leverage the power of very fast, low-level libraries making Python data-processing fast as well as easy.

This book aims to introduce some of those heavy-weight tools as well as a host of other smaller but equally vital tools. It also aims to show how Python and JavaScript in concert represent the best data-viz stack out there for anyone wishing to deliver their visualisations to the internet.

Up next is the first part of the book covering the preliminary skills needed for the toolchain. You can work through it now or skip ahead to part two and the start of the toolchain, referring back when ne

Recommended Books

Here's a few key data-visualisation books to whet your appetite, covering the gamut from interactive dashboards to beautiful and insightful info-graphics.

- The Visual Display of Quantitative Information. Edward Tufte. Graphics Press, 1983.
- Information Visualization: Perception for Design. Colin Ware. Morgan Kaufmann, 2004.
- Cartographies of Time: A History of the Timeline. Daniel Rosenberg. Princeton Architectural Press, 2012.

- Information Dashboard Design: Displaying Data for at-a-glance Monitoring. Stephen Few. Analytics Press, 2013.
- The Functional Art. Alberto Cairo. New Riders 2012.
- Semiology of Graphics: Diagrams, Networks, Maps. Jacques Bertin. Esri Press 2010.

CHAPTER 1

A Development Setup

This chapter will cover the downloads and software installations needed to use this book as well as sketching out a recommended development environment. As you'll see, this isn't as onerous as it might once have been. I'll cover Python and JavaScript dependencies separately and give a brief overview of cross-language IDEs.

Python

The bulk of the libraries covered in the book are Python-based but what might have been a challenging attempt to provide comprehensive installation instructions for the various Operating Systems and their quirks is made much easier by the existence of Continuum Analytics *Anaconda*, a Python platform which bundles together most of the popular analytics libraries in a convenient package.

Anaconda

Installing some of the bigger Python libraries used to be a challenge all in itself, particularly those such as Numpy which depend on complex low-level C and Fortran packages. That's why the existence of Anaconda is such a God-send. It does all the dependency check-

ing, binary installs etc. so you don't have to. It's also a very convenient resource for a book like this.

Python 2 or 3?

Right now Python is in transition to version 3, a process which is taking longer than many would like. This is because Python 2+ works fine for many people, a lot of code will have to be converted[1] and up until recently some of the big libraries, such as Numpy and Scipy, only worked for 3.

Now that most of the major libraries are Python 3 compatible it would be a no-brainer to recommend that version for this book. Unfortunately one of the few hold-outs, not yet v3. ready, is Scrapy, a big tool on our tool-chain[2] which you'll learn about in Chapter 6. I don't want to oblige you to run two versions so for that reason we'll be using the version 2. Anaconda package.

I will be using the new print function[3] which means all the non-Scrapy code will work fine with Python 3.

To get your free Anaconda install, just navigate your browser to *https://www.continuum.io/downloads*, choose the version for your Operating System (as of late 2015, the we're going with Python 2.7), and follow the instructions. Windows and OSX get a graphical installer (just download and double-click) while Linux requires you to run a little bash script:

```
$ bash Anaconda-2.3.0-Linux-x86_64.sh
```

I recommend sticking to defaults when installing Anaconda.

Checking the Anaconda install

The best way to check your Anaconda install went ok is to try firing up an IPython session at the command-line. How to do this depends on your operating system:

1 There are a number of pretty reliable automatic converters out there.

2 The Scrapy team are working hard to rectify this. Scrapy relies on Python's Twisted, an event-driven networking engine also making the journey to Python 3+ compatability.

3 This is imported from the *future* module, i.e. `from *future* import print_function.

At the Windows command-prompt:

```
C:\Users\Kyran>ipython
```

At the OS-X or Linux prompt:

```
$ ipython
```

This should produce something like the following:

```
kyran@Tweedledum:~/projects/pyjsbook$ ipython
Python 2.7.10 |Anaconda 2.3.0 (64-bit)|
               (default, May 28 2015, 17:02:03) Type
"copyright", "credits" or "license" for more information.

IPython 3.2.0 -- An enhanced Interactive Python.  Anaconda is
brought to you by Continuum Analytics.  Please check out:
http://continuum.io/thanks and
https://anaconda.org
...
```

Most installation problems will stem from a badly configured environment Path variable. This Path needs to contain the location of the main Anaconda Directory and its Scripts sub-directory. In Windows this should look something like:

```
'...C:\\Anaconda;C:\\Anaconda\Scripts...
```

You can access and adjust the environment variables in Windows 7 by typing "environment variables" in program's search field and selecting `Edit environment variables` or in XP via `Control Panel > System > Advanced > Environment Variables`.

In OS-X and Linux systems you should be able to set your `PATH` variable explicitly by appending this line to the `.bashrc` file in your home directory:

```
export PATH=/home/kyran/anaconda/bin:$PATH
```

Installing extra libs

Anaconda contains almost all the Python libraries covered in this book (see here for the full list of Anaconda libraries). Where we need a non-Anaconda library we can use *pip* (short for Pip Installs Python), the defacto standard for installing Python libraries. Using `pip` to install is as easy as can be, just call `pip install` followed by the name of the package from the command-line and it should be installed or, with any luck, give a sensible error:

```
$ pip install dataset
```

Virtual Environments

```
http://docs.python-guide.org/en/latest/dev/virtualenvs/[Virtual Environments] pro
```

Anaconda comes with a `conda` system command that makes creating and using virtual-environments easy. Let's create a special one for this book, based on the full Anaconda package:

```
$ conda create --name pyjsviz anaconda
...
#
# To activate this environment, use:
# $ source activate pyjsviz
#
# To deactivate this environment, use:
# $ source deactivate
#
```

As the final message says, to use this virtual environment you need only `source activate` it (for Windows machines you can leave out the `source`):

```
$ source activate pyjsviz
discarding /home/kyran/anaconda/bin from PATH
prepending /home/kyran/.conda/envs/pyjsviz/bin to PATH
(pyjsviz) $
```

Note that you get a helpful cue at the command-line to let you know which virtual environment you're using.

The `conda` command can do a lot more than just facilitate virtual environments, combining the functionality of Python's `pip` installer and `virtualenv` command, among other things. You can get a full run-down here.

JavaScript

The good news is that you don't need much JavaScript software at all. The only must-have is the Chrome/Chromium web-browser, which is used in this book. It offers the most powerful set of developer tools of any current browser and is cross-platform.

To download Chrome just go here and download the version for your operating system. This should be automatically detected.

If you want something slightly less Googlefied then you can use Chromium, the browser based on the open-source project from which Google Chrome is derived. You can find up to date instruc-

tions on installation here or just head on to the main download page. Chromium tends to lag Chrome feature-wise but is still an eminently usable development browser.

Content Delivery Networks (CDNs)

One of the reasons you don't have to worry about installing JavaScript libraries is that the ones used in this book are available via content delivery networks (CDNs). Rather than having the libraries installed on your local machine, the JavaScript is retrieved by the browser over the web, from the closest available server. This should make things very fast - faster than if you served the content yourself.

To include a library via CDN you use the usual `<script>` tag, usually placed at bottom of your HTML page. For example, his call adds the latest (as of late 2015) version of D3:

```
<script
 src="https://cdnjs.cloudflare.com/ajax/libs/d3/3.5.6/d3.min.js"
 charset="utf-8">
</script>
```

Installing libraries locally

If you need to install JavaScript libraries locally, e.g. you anticipate doing some offline development work or can't guarantee an internet connection, there are a number of fairly simple ways.

You can just download the separate libraries and put them in your local server's static folder. This is a typical folder-structure. Third-party libraries go in the `static/libs` directory off root, like so:

```
nobel_viz/
└── static
    ├── css
    ├── data
    ├── libs
    |     └── d3.min.js
    └── js
```

If you organise things this way, to use D3 in your scripts now requires a local file reference with the `<script>` tag:

```
<script src="/static/libs/d3.min.js"></script>
```

Databases

This book shows how to interact with the main SQL databases and MongoDB, the chief non-relational or NoSQL database, from Python. We'll be using SQLite, the brilliant file-based SQL database. Here's the download details for SQLite and MongoDB:

- SQLite - a great, file-based, serverless SQL-database. It should come as standard with Mac OS-X and Linux. For Windows, follow this guide.

- MongoDB - by a long way the most popular NoSQL database. Installation instructions here.

Note that we'll either be using Python's SQLAlchemy SQL-library directly or through libraries that build on it. This means any SQLite examples can be converted to another SQL backend (e.g. MySQL or PostgreSQL) by changing a configuration line or two.

Integrated Development Environments

As I explain in "The myth of IDEs, frameworks and tools" on page 108, I don't think you need an IDE to program in Python or JavaScript. The development tools provided by modern browsers, Chrome in particular, mean you only really need add a good code-editor to have pretty much the optimal setup. It's free as in beer too.

For Python, I have tried a few IDEs but they've never stuck. The main itch I was trying to scratch was a decent debugging system. Setting breakpoints etc. in Python with a text-editor isn't particularly elegant and using the command-line debugger `pdb` feels a little too old-school sometimes. Nevertheless, Python's logging etc. is so easy and effective that breakpoints were an edge-case which didn't justify leaving my favourite editor[4], which does pretty decent code-completion, solid syntax-highlighting etc..

In no particular order, here are a few I've tried and not disliked:

- PyCharm - solid code-assistance etc., good debugging.

4 Emacs with VIM key-bindings.

- PyDev - if you like Eclipse and can tolerate it's rather large footprint, this might well be for you.
- WingIDE is a solid bet, with a great debugger and incremental improvements over a decade and a half development.

Summary

With free, packaged Python distributions such as Anaconda and the inclusion of sophisticated Javscript development tools in freely available web-browsers, the necessary Python and JavaScript elements of your dev environment are a couple of clicks away. Add a favourite editor and a database of choice[1] and you are pretty much good to go. There are additional libraries such as node.js which can be useful but don't count as essential. Now we've established our programming environment, the next chapters will teach the preliminaries needed to start our journey of data-transformation along the toolchain. Starting with a language bridge between Python and JavaScript.

1 SQLite is great for development purposes and doesn't need a server running on your machine.

PART I

A Basic Toolkit

This first part of the book provides a basic toolkit for the toolchain to come and is part tutorial, part reference. Given the fairly wide range of knowledge in the book's target audience, there will probably be things covered that you already know. My advice is just to cherry-pick the material to fill any gaps in your knowledge and maybe skim what you already know as a refresher.

If you're confident you already have the basic toolkit to hand, feel free to skip to the start of our journey along the toolchain in Part II.

CHAPTER 2

A Language Learning Bridge Between Python and JavaScript

Probably the most ambitious aspect of this book is that it deals with two programming languages. Moreover, it only requires that you are competent in one of these languages. This is only possible because Python and JavaScript (JS) are fairly simple languages with much in common. The aim of this chapter is to draw out those commonalities and use them to make a learning-bridge between the two languages such that core skills acquired in one can easily be applied to the other.

After showing the key similarities and differences between the two languages I'll show how set up a learning environment for Python and JS. The bulk of the chapter will then deal with core syntactical and conceptual differences, followed by a selection of patterns and idioms that I use a lot while doing data visualisation work.

Similarities and differences

Syntax differences aside, Python and JavaScript actually have a lot in common. After a short while, switching between them can be

almost seamless[1]. Let's compare the two from a data-visualiser's perspective:

These are the chief similarities

- They both work without needing a compilation step (i.e. they are interpreted).
- You can use both with an interactive interpreter, which means you can type in lines of code and see the results right away.
- Both have garbage collection.
- Neither language has header files, package boilerplate etc..
- Both are primarily developed with a text-editor not an IDE.
- In both, functions are first class citizens which can be passed as arguments etc..

Their key differences

- Possibly the biggest difference is that JavaScript is single-threaded and non-blocking, using asynchronous I/O. This means simple things like file-access involve the use of a callback function.
- JS is used essentially in web-development, until very recently being browser bound[2] but Python is used almost everywhere.
- JS is the only first class language in web-browsers, Python being excluded.
- Python has a comprehensive standard library whereas JS a limited set of utility objects, e.g. JSON, Math.
- Python has fairly classical Object Oriented classes whereas JS uses prototypes.
- JS lacks general-purpose data-processing libs[3].

The differences here emphasise the need for this book to be bilingual. JavaScript's monopoly of browser dataviz needs the comple-

1 One particularly annoying little gotcha is that while Python uses `pop` to remove a list item, it uses `append` not `push` to add an item. Javscript uses `push` to add an item while `append` is used to concatenate arrays.

2 The ascent of node.js has extended JavaScript to the server.

3 This is changing with libraries like *crossfilter* but JS is far behind Python, *R* and others.

ment of a conventional data-processing stack. And Python has the best there is.

Interacting with the Code

One of the great advantages of Python and JavaScript is that because they are interpreted on-the-fly, you can interact with them. Python's interpreter can be run from the command-line while JavaScript's is generally accessed from the web-browser through a console, usually available from the in-built development tools. In this section we'll see how to fire up a session with the interpreter and start trying out your code.

Python

By far the best Python interpreter is *IPython*, which comes in three shades, the basic terminal version, an enhanced graphical version and a notebook. The notebook is a wonderful and fairly recent innovation, providing a browser-based interactive computational environment. There are pros and cons to the different versions. The command-line is fastest to scratch a problematic itch but lacks some bells and whistles, particularly embedded plotting courtesy of *Matplotlib* and friends. The makes it sub-optimal for *Pandas* based data-processing and data-visualisation work. Of the other two, both are better for multi-line coding (trying out functions etc.) than the basic interpreter but I find the graphical *qtconsole* more intuitive, having a familiar command-line rather than executable cells[4] . The great boon of the notebook is session persistence and the possibility of web-access[5]. The ease with which one can share programming sessions, complete with embedded data-visualisations, makes the notebook a fantastic teaching tool, as well as a great way to recover programming context.

You can start them at the command-line like this

```
$ ipython [qt | notebook]
```

options can be empty, for the basic command-line interpreter, *-qt* for a *Qt* based graphical version and *-notebook* for the browser-

4 This version is based on the Qt GUI library.

5 At the cost of a running a Python interpreter on the server.

based notebook. You can use any of the three IPython alternatives for this section but for serious interactive data-processing I generally find myself gravitating to the Qt console for sketches or the notebook if I anticipate an extensive project.

JavaScript

There are lots of options for trying out JavaScript code without starting a server, though the latter isn't that difficult. Because the JavaScript interpreter comes embedded in all modern web-browsers, there are a number of sites that let you try out bits of JavaScript along with HTML and CSS and see the results. *JSBin* is a good option. These sites are great for sharing code, trying out snippets etc. and usually allow you to add libraries such as D3.js.

If you want to try out code one-liners or quiz the state of live code, browser-based consoles are your best bet. With Chrome you can access the console with the key-combo `Ctrl-Shift-J`. As well as trying little JS snippets, the console allows you to drill-down into any objects in scope, revealing their methods and properties. This is a great way to quiz the state of a live object and search for bugs.

One disadvantage of using on-line JavaScript editors is losing the power of your favourite editing environment, with linting, familiar keyboard-shortcuts and the like (see Chapter 4). On-line editors tend to be rudimentary, to say the least. If you anticipate an extensive JavaScript session and want to use your favourite editor, the best bet is to run a local server.

First, create a project directory, called sandpit for example, and add a minimal HTML file which includes a JS script:

```
sandpit
├── index.html
└── script.js
```

The index.html file need only be a few lines long, with an optional `div` place-holder on which to start building your visualisation or just trying out a little DOM-manipulation.

```
<!-- index.html -->
<!DOCTYPE html>
<meta charset="utf-8">

<div id='viz'></div>
```

```
<script type="text/javascript" src="script.js" async></script>
```

You can then add a little JavaScript to your script.js file:

```
// script.js
var data = [3, 7, 2, 9, 1, 11];
var total = 0;
var sum = data.forEach(function(d){
    total += d;
});

console.log('Sum = ' + sum);
// outputs 'Sum = 33'
```

Start your development server in the project directory

```
$sandpit python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

Then open your browser at *http://localhost:8000*, press Ctrl-Shift-J (Cmd + Opt + J on a Mac) to access the console and you should see Figure 2-1, showing the logged output of the script (see Chapter 4 for further details).

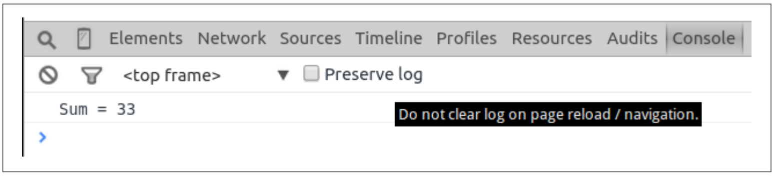


Figure 2-1. Outputting to the Chrome console

Now we've established how to run the demo code, let's start building a bridge between Python and JavaScript. First, we'll cover the basic differences in syntax. As you'll see, they're fairly minor and easily absorbed.

Basic Bridge Work

In this section I'll contrast the basic nuts and bolts of programming in the two languages.

Style guidelines, PEP 8 and 'use strict'

Where JavaScript style guidelines are a bit of a free for all (with people often defaulting to those used by a big library like jQuery), Python has a Python Enhancement Proposal (PEP) dedicated to it.

I'd recommend getting acquainted with PEP-8 but not submitting totally to its leadership. It's right about most things but there's room for some personal choice here. There's a handy on-line checker here which will pick up any infractions of PEP-8.

In Python you should use four spaces to indent a code-block. JavaScript is less strict but two spaces is the most common indent.

One recent addition to Javscript (Ecmascript 5) is the *use strict* directive, which imposes strict mode. This mode enforces some good Javscript practice, which includes catching accidental global declarations and I thoroughly recommend its use. To use it just place the string at the top of your function or module:

```
(function(foo){
  'use strict';
  // ...
}(window.foo = window.foo || {});
```

Camel-case vs underscore

JS conventionally uses camel-case (e.g. processStudentData) for its variables while Python, in accordance with PEP-8, uses underscores (e.g. process_student_data) in its variable names (Example 2-4 and Example 2-3 B). By convention (and convention is more important in the Python ecosystem than JS) Python uses capitalised camel-case for class declarations (see below), uppercase for constants and underscores for everything else:

```
FOO_CONST = 10
class FooBar(object): # ...
def foo_bar():
    baz_bar = 'some string'
```

Importing modules, including scripts

Using other libraries in your code, either your own or third party, is fundamental to modern programming. Which makes it all the more surprising that JavaScript doesn't really have a mechanism for doing it[6]. Python has a simple import system which, on the whole, works pretty well.

6 The constraint of having to deliver JS scripts over the web via HTTP is largely responsible for this.

The good news on the JavaScript front is that Ecmascript 6, the next version of the language, does address this issue, with the addition of `import` and `export` statements. Ecmascript 6 will be getting browser support soon but as of late 2015 you need a converter to Ecmascript 5 such as Babel.js. Meanwhile, although there have been many attempts to create a reasonable client-side modular system none have really achieved critical mass and all are a little awkward to use. For now I would recommend using the well-established HTML `script` tag to include scripts. So to include the D3 visualisation library you would add this tag to your main HTML file, conventionally `index.html`:

```
<!DOCTYPE html>
<meta charset="utf-8">
...
  <script src="http://d3js.org/d3.v3.min.js"></script>
```

You can include the script anywhere in your HTML file but it's best practice to add them after the body (div tags etc.) section[7] Note that the order of the script tags is important. If a script is dependent on a module, e.g. it uses the D3 library, its `script` tag must be placed after that of the module. i.e. big library scripts, such as jQuery and D3 will be included first.

Python comes with 'batteries included', a comprehensive set of libraries covering everything from extended data containers (`collections`) to working with the family of CSV files (`csv`). If you want to use one of these just import it using the `import` keyword:

```
In [1]: import sys

In [2]: sys.platform
Out[2]: 'linux2'
```

If you don't want to import the whole library, want to use an alias etc., you can use the `as` and `from` keywords instead:

```
import pandas as pd
from csv import DictWriter, DictReader
from numpy import * ❶

df = pd.read_json('data.json')
reader = DictReader('data.csv')
md = median([12, 56, 44, 33])
```

7 This means any blocking script loading calls occur after the page's HTML has rendered.

❶ This imports all the variables from the module into the current namespace and is almost always a bad idea. One of the variables could mask an existing one and it goes against Python best-practice of explicit being better than implicit. One exception to this rule is if you are using the Python interpreter interactively. In this limited context it may make sense to import all functions from a library to cut down on key-presses, e.g. importing all the math functions (`from math import *`) if doing some Python math hacking.

If you import a non-standard library, Python uses `sys.path` to try and find it. `sys.path` consists of:

- the directory containing the importing module (current directory)
- the `PYTHOPATH` variable, containing a list of directories
- the installation-dependent default, where libraries installed using `pip` or `easy_install` will usually be placed.

Big libraries are often packaged, being divided into sub-modules. These sub-modules are accessed by dot-notation:

```
import matplotlib.pyplot as plt
```

Packages are constructed from the filesystem using '*init*.py' files, usually empty, as shown in Example 2-1. The presence of an init file makes the directory visible to Python's import system.

Example 2-1. Building a Python package

```
mypackage
├── __init__.py
...
├── core
│   └── __init__.py
│   ...
...
└── io
    ├── __init__.py
    └── api.py
    ...
    └── tests
        └── __init__.py
        └── test_data.py
        └── test_excel.py ❶
```

```
        ...
...
```

❶ This module would be imported using `from mypackage.io.tests import test_excel`.

Packages on `sys.path` can be accessed from the root directory (that's mypackage in Example 2-1) using dot-notation. A special case of `import` is intra-package references. The *test_excel.py* submodule in Example 2-1 can import submodules from the mypackage package both absolutely and relatively:

```
from mypackage.io.tests import test_data ❶
from . import test_data ❷
import test_data ❷
from ..io import api ❸
```

❶ Imports the *test_data.py* module absolutely, from the package's head-directory.

❷ An explicit ('. import') and implicit relative import.

❸ A relative import from a sibling package of *tests*.

Keeping your namespaces clean

The variables defined in Python modules are encapsulated, which means that unless you import them explicitly, e.g. `from foo import baa`, you will be accessing them from the imported module's namespace using dot notation, e.g. `foo.baa`. This modularisation of the global namespace is quite rightly seen as a very good thing and plays to one of one of Python's key tenets, the importance of explicit statements over implicit. When analysing someone's Python code it should be possible to see exactly where a class, function or variable has come from. Just as importantly, preserving the namespace limits the chance of conflicting or masking variables - a big potential problem as code-bases get larger.

One of the main criticisms of JavaScript, and a fair one, is that it plays fast and loose with namespace conventions. The most egregious example of this is that variables declared outside of functions

or missing the ‘var’ keyword[8] are global rather than confined to the script in which they are declared. There are various ways to rectify this situation but the one I use and recommend is to make each of your scripts a self-calling function. This makes all variables declared using `var` local to the script/function, preventing them polluting the global namespace. Any objects, functions, variables etc. you want to make available to other scripts can be attached to an object which is part of the global namespace.

Example 2-2 demonstrates a module-pattern. The boilerplate head and tail (labelled 1. and 3.) effectively create an encapsulated module. This pattern is far from a perfect solution to modular JavaScript but is the best compromise I know until Ecmascript 6 and a dedicated import system becomes standard. One obvious disadvantage is that the module is part of the global namespace, which means, unlike in Python, there is no need to explicitly import it.

Example 2-2. A module pattern for JavaScript

```
(function(nbviz) { ❶
    'use strict';
    // ...
    nbviz.updateTimeChart = function(data) { ❷
    // ...
}(window.nbviz = window.nbviz || {})); ❸
```

❶ Receives the global `nbviz` object.

❷ Attaches the `updateTimeChart` method to the global `nbviz` object, effectively *exporting* it.

❸ If an `nbviz` object exists in the global (window) namespace pass it into the module function, otherwise add it to the global namespace.

Outputting ‘Hello World’

By far the most popular initial demonstration of any programming language is getting it to print or communicate ‘Hello World!’ in

8 This possibility of a missing ‘var’ can be removed by using the Ecmascript 5 *use strict* directive.

some form, so let's start with getting output from Python and JavaScript.

Python's output couldn't be much simpler but version 3 sees a change to the print statement, making it a proper function[9]

```
# In Python 2
print 'Hello World!'

# In Python 3
print('Hello World!')
```

You can use Python 3's print function in Python 2 by importing it from the *future* module:

```
from __future__ import print_function
```

If you're not using Python 3 then this is a sensible approach. The new print function is here to stay and it's best to get used to it now.

JavaScript has no print function but you can log output to the browser console:

```
console.log('Hello World!);
```

Simple data-processing

A good way to get an overview of the language differences is to see the same function written in both. Example 2-3 and Example 2-4 show a small, contrived example of data-munging in Python and Javscript respectively. We'll use these to compare Python and JS syntax.

Example 2-3. Simple Data-munging with Python

```
from __future__ import print_function

# A
student_data = [
    {'name': 'Bob', 'id':0, 'scores':[68, 75, 56, 81]},
    {'name': 'Alice', 'id':1,  'scores':[75, 90, 64, 88]},
    {'name': 'Carol', 'id':2, 'scores':[59, 74, 71, 68]},
    {'name': 'Dan', 'id':3, 'scores':[64, 58, 53, 62]},
]

# B
```

9 This is a good thing for reasons outlined in PEP 3105 here.

```python
def process_student_data(data, pass_threshold=60,
                         merit_threshold=75):
    """ Perform some basic stats on some student data. """

    # C
    for sdata in data:
        av = sum(sdata['scores'])/float(len(sdata['scores']))
        sdata['average'] = av

        if av > merit_threshold:
            sdata['assessment'] = 'passed with merit'
        elif av > pass_threshold:
            sdata['assessment'] = 'passed'
        else:
            sdata['assessment'] = 'failed'
        # D
        print("%s's (id: %d) final assessment is: %s"%(
          sdata['name'], sdata['id'], sdata['assessment'].upper()))

# E
if __name__ == '__main__':
    process_student_data(student_data)
```

Example 2-4. Simple data-munging with JavaScript

```javascript
// A (note deliberate and valid inconsistency in keys (some quoted
// and some unquoted)
var studentData = [
    {name: 'Bob', id:0, 'scores':[68, 75, 76, 81]},
    {name: 'Alice', id:1, 'scores':[75, 90, 64, 88]},
    {'name': 'Carol', id:2, 'scores':[59, 74, 71, 68]},
    {'name': 'Dan', id:3, 'scores':[64, 58, 53, 62]},
];

// B
function processStudentData(data, passThreshold, meritThreshold){
    passThreshold = typeof passThreshold !== 'undefined'? passThreshold: 60;
    meritThreshold = typeof meritThreshold !== 'undefined'? meritThreshold: 75;

    // C
    data.forEach(function(sdata){
        var av = sdata.scores.reduce(function(prev, current){
            return prev+current;
        },0) / sdata.scores.length;
        sdata.average = av;

        if(av > meritThreshold){
            sdata.assessment = 'passed with merit';
        }
        else if(av > passThreshold){
            sdata.assessment = 'passed';
```

```
        }
        else{
            sdata.assessment = 'failed';
        }
        // D
        console.log(sdata.name + "'s (id: " + sdata.id +
          ") final assessment is: " +
            sdata.assessment.toUpperCase());
    });

}

// E
processStudentData(studentData);
```

String construction

Section D in Example 2-4 and Example 2-3 show the standard way to print output to console or terminal. JavaScript has no `print` statement but will log to the browser's console through the `console` object.

```
console.log(sdata.name + "'s (id: " + sdata.id +
  ") final assessment is: " + sdata.assessment.toUpperCase());
```

Note that the integer variable *id* is coerced to a string, allowing concatenation. Python doesn't perform this implicit coercion so attempting to add a string to an integer in this way would give an error. Instead explicit conversion to string form is achieved using one of the `str` or `repr` functions.

In Section A Example 2-3 the output string is constructed using *C* type formatting. String (%s) and integer (%d) place-holders are provided by a final tuple (%(…)):

```
print("%s's (id: %d) final assessment is: %s"
  %(sdata['name'], sdata['id'], sdata['assessment'].upper()))
```

These days I rarely use Python's `print` statement, opting for the much more powerful and flexible `logging` module, which is demonstrated in the following code-block. It takes a little more effort to use but it is worth it. Logging gives you the flexibility to direct output to a file and/or the screen, adjust the logging level to prioritise certain information and a whole load of other useful things. Check out the details here.

```
import logging
logger = logging.getLogger(__name__) ❶
```

```
//...
logger.debug('Some useful debugging output')
logger.info('Some general information')

// IN INITIAL MODULE
logging.basicConfig(level=logging.DEBUG) ❷
```

❶ Creates a logger with the name of this module.

❷ You can set the logging level, an output file as opposed to the default to screen etc..

Significant whitespace vs curly brackets

The syntactic feature most associated with Python is significant whitespace. Wheras languages like C and JavaScript use whitespace for readability and could easily be condensed into one line[10], in Python leading spaces are used to indicate code-blocks and removing them changes the meaning of the code. The extra effort required to maintain correct code alignment is more than compensated for by increased readability - you spend far longer reading than writing code and the easy reading of Python is probably the main reason why the Python library ecosystem is so healthy. Four spaces is pretty much mandatory (see PEP 8) and my personal preference is for what is know as *soft tabs*, where your editor inserts (and deletes) multiple spaces instead of a tab character[11]

In the following code, the indentation of the `return` statement must be four spaces by convention[12].

```
def doubler(x):
    return x * 2
#  |<-this spacing is important
```

JavaScript doesn't care about the number of spaces between statements, variables etc.., using curly-brackets to demark code-blocks, the two doubler functions in this code being equivalent:

10 this is actually done by JavaScript compressors to reduce the filesize of downloaded web-pages

11 The soft vs hard tab debate generates controversy, with much heat and little light. *PEP 8* stipulates spaces, which is good enough for me.

12 It could be two or even three spaces but this number must be consistent throughout the module.

```
var doubler = function(x){
  return x * 2;
}

var doubler=function(x){return x*2;}
```

Much is made of Python's whitespace but most good coders I know set their editors up to enforce indented code-blocks and a consistent look and feel. Python merely enforces this good practice. And, to reiterate, I believe the extreme readability of Python code contributes as much to Python's supremely healthy ecosystem as its simple syntax.

Comments and doc-strings

To add comments to code, Python uses hashes #:

```
# ex.py, a single informative comment

data = {} # Our main data-ball
```

By contrast, JavaScript uses the *C* language convention of double backslashes // or /* … */ for multi-line comments:

```
// script.js, a single informative comment
/* A multi-line comment block for
function descriptions, library script
headers and the like */
var data = {}; // Our main data-ball
```

As well as comments, and in keeping with its philosophy of readability and transparency, Python has documentation strings (docstrings) by convention. The *process_student_data* function in Example 2-3 has a triple-quoted line of text at its top which will automatically be assigned to the function's `__doc__` attribute. You can also use multi-line docstrings.

```
def doubler(x):
    """This function returns double its input."""
    return 2 * x

def sanitize_string(s):
    """This function replaces any string spaces
    with '-' after stripping any white-space
    """
    return s.strip().replace(' ', '-')
```

Docstrings are a great habit to get into, particularly if working collaboratively. They are understood by most decent Python editing

toolsets and are also used by such automated documentation libraries as *Sphinx*. The string-literal docstring is accessible as the *doc* property of a function or class.

Declaring variables, var

In Section A of Example 2-3 and Example 2-4 the declaration of the student data requires a `var` keyword for JavaScript. We could dispense with the `var` and the script would run fine but we would be in danger of being skewered by JS gotcha number one: any variables declared without `var` are attached to the global namespace, or window object, which means they can easily mask or be masked by any other variables sharing the same name. This possibility of namespace pollution is a big problem for JS and the reason why you should get a good linter to warn of missing `var`s. You should also use Ecmascript's *use strict* directive to force all variables to be declared with `var` (see "Style guidelines, PEP 8 and 'use strict'" on page 37).

Strictly speaking JS statements should be terminated with a semi-colon as opposed to Python's new line. You will see examples where the semi-colon is dispensed with and modern browsers will usually do the right thing here but there are risks involved (e.g. it can trip up code minifiers and compressors which remove white-space). I'm in the semi-colon camp but many smart people seem to make do without them.

> Declare all variables to be used in a function at its top. Javscript has *variable hoisting*, which means variables are processed before any other code. This means declaring them anywhere in the function is equivalent to declaring them at the top. This can result in weird errors and confusion. Explicitly placing vars at the top avoids this.

Strings and numbers

The *name* strings used in the student-data (see Section A of Example 2-3 and Example 2-4) will be interpreted as *UCS-2* (the

parent of unicode *UTF-16*) in JavaScript[13], a string of bytes in Python 2 and unicode (*UTF-8* by default) in Python 3[14]

Both languages allow single and double quotes for strings. If you want to include a single or double quote in the string then enclose with the alternative, like so:

```
pub_name = "The Brewer's Tap"
```

The *scores* in Section A Example 2-4 are stored as JavaScript's one numeric type, double-precision 64-bit (IEEE 754) floating-point numbers. Although JavaScript has a `parseInt` conversion function, when used with float's[15] it is really just a rounding operator, similar to `floor`. The type of the parsed `number` is still `number`:

```
var x = parseInt(3.45); // 'cast' x to 3
typeof(x); // "number"
```

Python has three numeric types, the 32 bit `int`, to which the student scores are cast, a `float` equivalent (IEE 754) to JS's `number` and a `long` for arbitrary precision integer arithmetic. This means Python can represent any integer whereas JavaScript is more limited[16]. Python's casting changes type:

```
foo = 3.4 # type(foo) -> float
bar = int(3.4) # type(bar) -> int
```

The nice thing about Python and JavaScript numbers is that they are easy to work with and usually do what you want. If you need something more efficient, Python has the *Numpy* library which allows fine-grained control of your numeric types (you'll learn more about Numpy in ???. In JavaScript, aside from some cutting edge projects, you're pretty much stuck with 64 bit floats.

13 The quite fair assumption that JavaScript uses UTF-16 has been the cause of much bug-driven misery. See here for an interesting analysis.

14 The change to unicode strings in Python 3 is a big one. Given the confusion that often attends unicode de/encoding it's worth reading a little bit about it: *https://docs.python.org/3/howto/unicode.html*

15 `parseInt` can do qutie a bit more than round. For example parseInt(*12.5px*) gives 12, first removing the *px* and then casting the string to a number. It also has a second *radix* argument to specify the base of the cast. See here for the specifics.

16 With very large numbers JavaScript can get very weird, with non-continuous integer.

Booleans

Python differs from the JavaScript and the *C* class languages in using named booleans operators. Other than that they work pretty much as expected. This table gives a comparison:

Python	bool	True	False	not	and	or
JavaScript	boolean	true	false	!	&&	$$

Python's capitalised *True* and *False* is an obvious trip up for any JavaScripter and vice-versa but any decent syntax-highlighting should catch that as should your code-linter.

Rather than always returning boolean true or false, both Python and JavaScript `and`/`or` expressions return the result of one of the arguments, which may of course be a boolean value. The following table shows how this works, using Python to demonstrate:

Table 2-1. Pythons' boolean operators

Operation	Result
x or y	if x is false, then y, else x
x and y	if x is false, then x, else y
not x	if x is false, then True, else False

This fact allows for some occasionally useful variable assignments

```
rocket_launch = True
(rocket_launch == True and 'All OK') or 'We have a problem!'
Out:
'All OK'

rocket_launch = False
(rocket_launch == True and 'All OK') or 'We have a problem!'
Out:
'We have a problem!'
```

Data containers: dicts, objects, lists, arrays

Roughy speaking, Javscript `objects` can be used like Python `dicts` and Python `lists` like JavaScript arrays. Python also has an tuple

container, which functions like an immutable list. Here's some examples:

```
# Python
d = {'name': 'Groucho', 'occupation': 'Ruler of Freedonia'}
l = ['Harpo', 'Groucho', 99]
t = ('an', 'immutable', 'container')

// JavaScript
d = {'name': 'Groucho', 'occupation': 'Ruler of Freedonia'}
l = ['Harpo', 'Groucho', 99]
```

As shown in Section A Example 2-3 and Example 2-4, while Python's dict keys must be quote-marked strings (or hashable types), JavaScript allows you to omit the quotes if the property is a valid identifier, i.e. not containing special characters such as spaces, dashes etc.. So in our studentData objects JS implicitly converts the property *name* to string form.

The student data declarations look pretty much the same and, in practice, are used pretty much the same too. The key difference to note is that while the curly-bracketed containers in the JS `studentData` look like Python dicts, they are actually a shorthand declaration of JS objects, a somewhat different data-container.

In JS data-visualisation we tend to use arrays of objects as the chief data-container and here JS objects function much as a Pythonista would expect. In fact, as demonstrated in the following code, we get the advantage of both dot notation and key-string access, the former being preferred where applicable (keys with spaces, dashes etc. needing quoted strings):

```
var foo = {bar:3, baz:5};
foo.bar; // 3
foo['baz']; // 5, same as Python
```

It's good to be aware that though they can be used like Python dictionaries, JavaScript objects are much more than just containers (aside from primitives like strings and numbers pretty much everything in Javscript is an object)[17]. But in most dataviz examples you see, they are used very much like Python `dicts`.

Here's a little table to convert basic list operations:

17 This makes iterating over their properties a little trickier than it might be. See here for more details.

Table 2-2. Lists and arrays

JavaScript array (a)	Python list (l)
a.length	len(l)
a.push(item)	l.append(item)
a.pop()	l.pop()
a.shift()	l.pop(0)
a.unshift(item)	l.insert(0, item)
a.slice(start, end)	l[start:end]
a.splice(start, howMany, i1, ...)	l[start:end] = [i1, ...]

Functions

Section *B* of Example 2-3 and Example 2-4 shows a function declaration. Python uses *def* to indicate a function:

```
def process_student_data(data, pass_threshold=60,
                         merit_threshold=75):
    """ Perform some basic stats on some student data. """
    ...
```

Whereas JavaScript uses *function*:

```
function processStudentData(data, passThreshold, meritThreshold){
    passThreshold = typeof passThreshold !== 'undefined'? passThreshold: 60;
    meritThreshold = typeof meritThreshold !== 'undefined'? meritThreshold: 75;
    ...
}
```

Both have a list of parameters. With JS the function codeblock is indicated by the curly brackets *{ ... }*, with Python the code-block is defined by a colon and indentation.

JS has an alternative way of defining a function, the function expression, which you may see in examples:

```
var processStudentData = function( ...){
```

The differences are subtle enough not to worry now[18]. For what it's worth, I tend to use function expressions pretty much exclusively.

Function parameters is an area where Python's handling is a deal more sophisticated than JavaScript's. As you can see in `process_student_data` (Section B Example 2-3), Python allows default arguments for the parameters. In JavaScript all parameters not used in the function call are declared *undefined*. In order to set a default value for these we have to perform a distinctly hacky conditional (ternary) expression:

```
function processStudentData(data, passThreshold, meritThreshold){
    passThreshold = typeof passThreshold !== 'undefined'? passThreshold: 60;
    ...
----
```

The good news for JavaScripters is that the latest version of JavaScript, based on Ecmascript 6 and coming very soon allows Python-like default parameters:

```
function processStudentData(data, passThreshold = 60, meritThreshold = 75){
...
```

Iterating: for loops and functional alternatives

Section C Example 2-4 and Example 2-3 shows our first major departure, demonstrating JavaScript's functional chops.

Python's for loops are simple, intuitive and work on any iterator[19], for example arrays and dicts. One gotcha with dicts is that standard iteration is by key, not items. For example:

```
foo = {'a':3, 'b':2}
for x in foo:
    print(x)
# outputs 'a' 'b'
```

To iterate over the key-value pairs, use the *dict*'s *items* method like so:

```
for x in foo.items():
    print(x)
# outputs key-value tuples ('a', 3) ('b' 2)
```

18 For the curious, there's a nice summation here.

19 see below for generators and pseudo containers

You can assign the key/values in the for statement for convenience. For example:

```
for key, value in foo.items():
```

Because Python's for loop works on anything with the correct iterator plumbing, you can do cool things like loop over file lines:

```
for line in open('data.txt'):
    print(line)
```

Coming from Python, JS's for loop is a pretty horrible, unintuitive thing. Here's an example:

```
for(var i in ['a', 'b', 'c']){
  console.log(i)
}
# outputs 1, 2, 3
```

JS's *for .. in* returns the index of the array's items, not the items themselves. To compound matters, for the *Pythonista*, the order of iteration is not guaranteed, so the indexes could be returned in non-consecutive order.

Even iterating over an object is trickier than it might be. Unlike Python's dicts, objects could have inherited properties from the prototyping chain so you should use a *hasOwnProperty* guard to filter these out, like so:

```
var obj = {a:3, b:2, c:4};
for (var prop in obj) {
  if( obj.hasOwnProperty( prop ) ) {
    console.log("o." + prop + " = " + obj[prop]);
  }
}
// out: o.a = 3, o.b = 2, o.c = 4
```

Shifting between Python and JS for loops is hardly seamless, demanding you keep on the ball. The good news is that you hardly need to use JS for-loops these days. In fact, I almost never find the need. That's because JS has recently acquired some very powerful first-class functional abilities, which have more expressive power, less scope for confusion with Python and, once you get used to them, quickly become indispensable[20].

20 this is one area where JS beats Python hands-down and which finds many of us wishing for similar functionality in Python.

Section C Example 2-4 demonstrates *forEach()*, one of the *functional* methods available to modern JavaScript arrays[21]. *forEach()* iterates over the array's items, sending them in turn to an anonymous callback function defined in the first argument where they can be processed. The true expressive power of these functional methods comes from chaining them (maps, filters etc.) but already we have a cleaner, more elegant iteration with none of the awkward bookkeeping of old.

The callback function receives index and the original array as optional second argument

```
data.forEach(function(currentValue, index){//---
```

Whereas JS arrays have a set of native functional iterator methods (map, reduce, filter, every, sum, reduceRight), `Objects` -in their guise as pseudo-dictionaries- don't. If you want to iterate over `Object` key-value pairs then I'd recommend using underscore[22], the most used functional library for JS and almost as ubiquitous as jQuery. Underscore methods are accessed with the shorthand _, like this:

```
_.each(obj, function(value, key){
    // do something with the data..
```

This does introduce a library dependency but this type of iteration is very common in data-visualisation work and underscore has lots of other goodies. Along with jQuery it has pretty much honorary JS standard-library status.

Conditionals: if, else, elif, switch

Section C Example 2-3 and Example 2-4 shows Python and JavaScript conditionals in action. Aside from JavaScript's bracket fetish, the statements are very similar, the only real difference being Python's extra `elif` keyword, a convenient conjunction of `else if`.

Though much requested, Python does not have the *switch* statement found in most high-level languages. JS does, allowing you to do this:

```
switch(expression){
  case value1:
```

21 Added with Ecmascript 5 and available on all modern browsers.

22 I use lodash, which is functionally identical

```
      // execute if expression === value1
      break; // optional end expression
    case value2:
      //...
    default:
      // if other matches fail
```

File input and output

JavaScript has no real equivalent of file input and output (I/O) but Python's is as simple as could be:

```
# READING A FILE
f = open("data.txt") # open file for reading

for line in f: # iterate over file-lines
    print(line)

lines = f.readlines() # grab all lines in file into array
data = f.read() # read all of file as single string

# WRITING TO A FILE
f = open("data.txt", 'w') # use 'w' to write, 'a' to append to file
f.write("this will be written as a line to the file")
f.close() # explicitly close the file
```

One much recommended best-practice is to use Python's `with`, `as` context manager when opening files. This ensures they are closed automatically when leaving the block, essentially providing syntactic sugar for a try, except, finally block. Here's how to open a file using with, as:

```
with open("data.txt") as f:
    lines = f.readlines()
    ...
```

Classes and prototypes

Possibly the cause of more confusion that any other topic is JavaScript's choice of prototypes rather than classical classes as its chief Object Orientated Programming (OOP) element. I have come to appreciate the concept of prototypes, if not its JS implementation, which could have been cleaner. Nevertheless, once you get the basic principle you may find that it is actually a better mental model for much of what we do as programmers than classical OOP paradigms.

I remember, when I first started my forays into more advanced languages like C++, falling for the promise of OOP, particularly class-

based inheritance. Polymorphism was all the rage and Shape classes were being sub-classed to rectangles and ellipses, which were in turn subclassed to more specialised squares and circles.

It didn't take long to realise that the clean class divisions found in the text-books were rarely found in real programming and that trying to balance generic and specific APIs quickly became fraught. In this sense, I find composition and mix-ins much more useful as a programming concept than attempts at extended subclassing and often avoid all these by using functional programming techniques, particularly in JavaScript. Nevertheless, the class/prototype distinction is an obvious difference between the two languages and the more you understand its nuances the better you'll code[23]

Python's classes are fairly simple affairs and, like most of the language, easy to use. I tend to think of them these days as a handy way to encapsulate data with a convenient API and rarely extend subclassing beyond one generation. Here's a simple example:

```
class Citizen(object):

    def __init__(self, name, country): ❶
        self.name = name
        self.country = country

    def print_details(self):
        print('Citizen %s from %s'%(self.name, self.country))


c = Citizen('Groucho M.', 'Freedonia') ❷
c.print_details()
Out:
Citizen Groucho M. from Freedonia
```

❶ Python classes have a number of double-underscored special methods, `__init__` being the most common, called when the class instance is created. All instance methods have a first, explicit *self* argument (you could name is something else but it's

23 I mentioned to a talented programmer friend that I was faced with the challenge of explaining prototypes to Python programmers and he pointed out that most JavaScripters could probably do with some pointers too. There's a lot of truth in this and many JSers do manage to be productive by using prototypes in a *classy* way, hacking their way around the edge-cases.

a very bad idea) which refers to the instance. In this case we use it to set name and country properties.

❷ Creates a new Citizen instance, initialised with name and country.

Python follows a fairly classical pattern of class inheritance. It's easy to do, which is probably why Pythonistas make a lot of use of it. Let's customise the `Citizen` class to create a (Nobel) `Winner` class with a couple of extra properties:

```
class Winner(Citizen):

    def __init__(self, name, country, category, year):
        super(Winner, self).__init__(name, country) ❶
        self.category = category
        self.year = year

    def print_details(self):
        print('Nobel winner %s from %s, category %s, year %s'\
        %(self.name, self.country, self.category, str(self.year)))


w = Winner('Albert E.', 'Switzerland', 'Physics', 1921)
w.print_details()
Out:
Nobel prize-winner Albert E. from Switzerland, category Physics,
year 1921
```

❶ We want to reuse the super-class `Citizen`'s `__init__` method, using this `Winner` instance as `self`. The `super` method scales the inheritance tree one branch from its first argument, supplying the second as instance to the class-instance method.

I think the best article I have read on the key difference between JavaScript's prototypes and classical classes is Reginald Braithwaite's "OOP, JavaScript, and so-called Classes". This quote sums up the difference between classes and prototypes as nice as any I've found:

> "The difference between a prototype and a class is similar to the difference between a model home and a blueprint for a home."

When you instantiate a C++ or Python class, a blueprint is followed, creating an object and calling its various constructors in the inheritance tree. In other words you start from scratch and build a nice, pristine new class instance.

With JavaScript prototypes you start with a model home (object) which has rooms (methods). If you want a new living room you can just replace the old one with something in better colors etc. If you want a new conservatory then just make an extension. But rather than building from scratch with a blueprint, you're adapting and extending an existing object.

With that necessary theory out of the way and the reminder that object-inheritance is useful to know but hardly ubiquitous in dataviz, let's see a simple JavaScript prototype example, Example 2-5.

Example 2-5. A simple Javscript object

```
var Citizen = function(name, country){ ❶
  this.name = name; ❷
  this.country = country;
};

Citizen.prototype = { ❸
  printDetails: function(){
    console.log('Citizen ' + this.name + ' from ' + this.country);
  }
};

var c = new Citizen('Groucho M.', 'Freedonia'); ❹

c.printDetails();
Out:
Citizen Groucho M. from Freedonia

typeof(c) # object
```

❶ Javscript has no classes[24] so object-instances are built from functions or objects.

❷ `this` is an implicit reference to the *calling context* of the function. For now it behaves as you would expect but though it looks a little like Python's `self` the two are quite different, as we'll see.

24 As of Ecmascript 6 this will change with the addition of the `class` keyword, a piece of syntactic sugar generating a lot of heat and not much light right now.

❸ The methods specified here will both override any prototypical methods up the inheritance chain and be inherited by any objects derived from `Citizen`.

❹ `new` is used to create a new object, set its prototype to the `Citizen` function and then call

self vs this

It would be easy enough, at first glance, to assume that Python's `self` and Javscript's `this` are essentially the same, the latter being an implicit version of the former, which is supplied to all class instance methods. But actually `this` and `self` are significantly different. Let's use our bi-lingual `Citizen` class to demonstrate.

Python's `self` is a variable supplied to each class method, (you can call it anything you like but it's not advisable) representing the class instance. But `this` is a keyword that refers to the object calling the method. This calling object can be different from the method's object instance and Javascript provides the `call`, `bind` and `apply` function methods to allow you to exploit this fact.

Let's use the `call` method to change the calling object of a `print_details` method and therefore the reference for `this`, used in the method to get the Citizen's name:

```
var groucho = new Citizen('Groucho M.', 'Freedonia');
var harpo = new Citizen('Harpo M.', 'Freedonia');

groucho.print_details.call(harpo);
Out:
"Citizen Harpo M. from Freedonia"
```

So JavaScript's `this` is a much more malleable proxy than Python's `self`, offering more freedom but also the responsibility of tracking calling context and, should you use it, making sure `new` is always used in creating objects[25].

I included Example 2-5 which shows `new` in JavaScript object instantiation because you will run into its use a fair deal. But the syntax is

25 This is another reason to use Ecmascript 5's *use strict;* injunction, which calls attention to such mistakes.

already a little awkward and gets quite a bit worse when you try to do inheritance. Ecmascript 5 introduced the `Object.create` method, a better way to create objects and to implement inheritance. I'd recommend using it in your own code but `new` will probably crop up in some third party libraries.

Let's use `Object.create` to create a `Citizen` and its `Winner` inheritor. To emphasise, Javscript has many ways to do this but Example 2-6 shows the cleanest I have found and my personal pattern.

Example 2-6. Prototypical inheritance with Object.create

```
var Citizen = { ❶
    setCitizen: function(name, country){
        this.name = name;
        this.country = country;
        return this;
    },
    printDetails: function(){
        console.log('Citizen ' + this.name + ' from ', + this.country);
    }
};

var Winner = Object.create(Citizen);

Winner.setWinner = function(name, country, category, year){
    this.setCitizen(name, country);
    this.category = category;
    this.year = year;
    return this;
};

Winner.printDetails = function(){
    console.log('Nobel winner ' + this.name + ' from ' +
    this.country + ', category ' + this.category + ', year ' +
    this.year);
};

var albert = Object.create(Winner)
    .setWinner('Albert Einstein', 'Switzerland', 'Physics', 1921);

albert.printDetails();
Out:
Nobel winner Albert Einstein from Switzerland, category
Physics, year 1921
```

❶ Citizen is now an Object rather than a constructor function. Think of this as the base-house for any new buildings such as Winner.

To reiterate, prototypical inheritance is not seen that often in JavaScript data-viz, particularly the 800-pound gorilla D3, with its emphasis on declarative and functional patterns, with *raw* unencapsulated data being used to stamp its impression on the web-page.

The tricky class/prototype comparison concludes this section on basic syntactic differences. Now let's look at some common patterns seen in dataviz work with Python and JS.

Differences in Practice

The syntactic differences between JS and Python are important to know and thankfully outweighed by their syntactic similarities. The meat and potatoes of imperative programming, loops, conditionals, data declaration and manipulation is much the same. This is all the more so in the specialised domain of data-processing and data-visualisation where the languages first class functions allow common idioms.

What follows is a less than comprehensive list of some important patterns and idioms seen in Python and JavaScript, from the perspective of a data-visualiser. Where possible a translation between the two languages.

Method chaining

A common JavaScript idiom is method chaining, popularised by its most popular library, jQuery and much used in D3. Method chaining involves returning an object from its own method in order to call another method on the result, using dot-notation:

```
var sel = d3.select('#viz')
    .attr('width', '600px') ❶
    .attr('height', '400px')
    .style('background', 'lightgray');
```

❶ The `attr` method returns the D3 selection that called it, which is then used to call another `attr` method.

Method chaining is not much seen in Python, which generally advocates one statement per line, in keeping with simplicity and readability.

Enumerating a list

Often it's useful to iterate through a list while keeping track of the item's index. Python has the very handy `enumerate` keyword for just this reason:

```
names = ['Alice', 'Bob', 'Carol']

for i, n in enumerate(names):
    print('%d: %s'%(i, n))

Out:
0: Alice
1: Bob
2: Carol
```

JavaScript's list methods, such as `forEach` and the functional `map`, `reduce` and `filter`, supply the iterated item and its index to the callback function:

```
var names = ['Alice', 'Bob', 'Carol'];

names.forEach(function(n, i){
    console.log(i + ': ' + n);
});

Out:
0: Alice
1: Bob
2: Carol
```

Tuple unpacking

One of the first cool tricks Python initiates come across uses tuple unpacking to switch variables:

```
(a, b) = (b, a)
```

Note that the brackets are optional. This can be put to more practical purpose as a way of reducing the temporary variables, for example in a fibonacci function:

```
def fibonacci(n):
    x, y = 0, 1
    for i in range(n):
```

```python
        print(x)
        x, y = y, x + y
```

If you want to ignore one of the unpacked variables, use an underscore:

```python
winner = 'Albert Einstein', 'Physics', 1921, 'Swiss'

name, _, _, nationality = winner
```

Tuple unpacking has a slew of use-cases. It is also a fundamental feature of the language and not available in JavaScript.

Collections

One of the most useful Python 'batteries' is the collections module. This provides some specialised container datatypes to augment Python's standard set. It has a `deque`, which provides a list-like container with fast appends and pops at either end, an `OrderedDict` which remembers the order entries were added, a `defaultdict`, which provides a factory function to set the dictionary's default and a `Counter` container for counting hashable objects, among others. I find myself using the last three a lot. Here's a few examples:

```python
from collections import Counter, defaultdict, OrderedDict

items = ['F', 'C', 'C', 'A', 'B', 'A', 'C', 'E', 'F']

cntr = Counter(items)
print(cntr)
cntr['C'] -=1
print(cntr)
Out:
Counter({'C': 3, 'A': 2, 'F': 2, 'B': 1, 'E': 1})
Counter({'A': 2, 'C': 2, 'F': 2, 'B': 1, 'E': 1})


d = defaultdict(int) ❶

for item in items:
    d[item] += 1 ❷

d
Out:
defaultdict(<type 'int'>, {'A': 2, 'C': 3, 'B': 1, 'E': 1, 'F': 2})


OrderedDict(sorted(d.items(), key=lambda i: i[1])) ❸
```

```
Out:
OrderedDict([('B', 1), ('E', 1), ('A', 2), ('F', 2), ('C', 3)]) ❹
```

❶ Sets the dictionary default to an integer, value 0 by default.

❷ If the item-key doesn't exist its value is set to the default of zero and 1 added to that.

❸ Gets the list of items in the dictionary `d` as key-value tuple pairs, sorts using the integer value and then creates an `OrderedDict` with the sorted list.

❹ The `OrderedDict` remembers the (sorted) order of the items as they were added to it.

You can get more details on the `collection` module from here.

There is a recent JavaScript library that emulates the Python `collections` module. You can find it here. As of late 2015 it is a very new but impressive piece of work, worth checking out even if you just want to extend your JavaScript knowledge.

If you want to replicate some of Python's `collection` function using more conventional JavaScript libraries, underscore or its functionally identical replacement lodash[26] are a good place to start. These libraries offer some enhanced functional programming utilities. Let's take a quick look at these handy tools.

Underscore

Underscore is probably the most popular JavaScript library after the ubiquitous jQuery and offers a bevy of functional programming utilities for the JavaScript dataviz programmer. The easiest way to use underscore is to use a content delivery network (CDN) to load it remotely (these loads will be cached by your browser, making things very efficient for common libraries), like so:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/
                underscore.js/1.8.3/underscore-min.js"></script>
```

26 My personal choice for performance reasons.

Underscore has loads of useful functions. There is, for example, a `countBy` method which serves the same purpose as the Python's collections Counter just discussed:

```
var items = ['F', 'C', 'C', 'A', 'B', 'A', 'C', 'E', 'F'];

_.countBy(items) ❶
Out:
Object {F: 2, C: 3, A: 2, B: 1, E: 1}
```

❶ Now you see why the library is called underscore.

As we'll now see, the inclusion in modern JavaScript of native functional methods (map, reduce, filter) and a `forEach` iterator for arrays has made underscore slightly less indispensable but it still has some great utilities to augment vanilla JS. With a little chaining you can produce extremely terse but very powerful code. Underscore was my gateway drug to functional programming in JavaScript and the idioms are just as addictive today. Check out underscores repertoire of utilities here.

Let's have a look at underscore in action, tackling a more involved task:

```
journeys = [
  {period:'morning', times:[44, 34, 56, 31]},
  {period:'evening', times:[35, 33],},
  {period:'morning', times:[33, 29, 35, 41]},
  {period:'evening', times:[24, 45, 27]},
  {period:'morning', times:[18, 23, 28]}
];

var groups = _.groupBy(journeys, 'period');
var mTimes = _.pluck(groups['morning'], 'times');
mTimes = _.flatten(mTimes); ❶
var average = function(l){
  var sum = _.reduce(l, function(a,b){return a+b},0);
  return sum/l.length;
};
console.log('Average morning time is ' + average(mTimes));
Out:
Average morning time is 33.81818181818182
```

❶ Our array of morning times arrays ([[44, 34, 56, 31], [33…]]) needs to be *flattened* into a a single array of numbers.

Functional array methods and list comprehensions

I find myself using underscore a lot less since the addition, with Ecmascript 5, of functional methods to JavaScript arrays. I don't think I've used a conventional for-loop since then which, given the ugliness of JS for-loops, is a very good thing.

Once you get used to processing arrays functionally, it's hard to consider going back. Combined with JS's anonymous functions it makes for very fluid, expressive programming. It's also an area where method chaining seems very natural. Let's look at a highly contrived example:

```
var nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

var sum = nums.filter(function(o){ return o%2 }) ❶
  .map(function(o){ return o * o}) ❷
  .reduce(function(a, b){return a+b}); ❸

console.log('Sum of the odd squares is ' + sum);
```

❶ Filters the list for odd numbers, i.e. returning 1 for the modulus (%) 2 operation.

❷ `map` produces a new list by applying a function to each member, i.e. [1, 3, 5…] → [1, 9, 25…].

❸ `reduce` processes the resultant mapped list in sequence, providing the current (in this case summed) value (a) and the item value (b). By default, the initial value of the first argument (a) is 0.

Python's powerful list comprehensions can emulate the example above easily enough:

```
nums = range(10) ❶

odd_squares = [x * x for x in nums if x%2] ❷
sum(odd_squares) ❸
Out:
165
```

❶ Python has a handy built-in `range` keyword, which can also take a start, end and step, e.g. range(2, 8, 2) → [2, 4, 6]

❷ The `if` condition tests for oddness of x and any numbers passing this filter are squared and inserted into the list.

❸ Python also has a built in and often used `sum` statement.

.

> Python's list comprehensions can use recursive control structures, applying a second for/if expression to the iterated items etc. Although this can create terse and powerful lines of code it goes against the grain of Python's readability and I would discourage its use. Even simple list-comprehensions are less than intuitive and as much as it appeals to the leet hacker in all of us, you risk creating incomprehensible code.

Python's list comprehensions work well for basic filtering and mapping. They do lack the convenience of JavaScript's anonymous functions (which are fully fledged, with their own scope, control blocks, exception handling etc.) but there are arguments against the use of anonymous functions. For example, they are not reusable and, being unnamed, they make it hard to follow exceptions and debug. See here for some persuasive arguments. Having said that, for libraries like D3, replacing the small, throw-away anonymous functions used to set DOM attributes and properties with named ones would be far too onerous and just add to the boilerplate.

Python does have functional lambda expressions, which we'll look at in the next section, but for full functional processing in Python by necessity and Javscript for best-practice, we might use named functions to increase our control scope. For our simple odd-squares example named functions are a contrivance but note that they increase the first-glance readability of the list comprehension - much more important as your functions get more complex.

```
def is_odd(x):
    return x%2

def sq(x):
    return x * x

sum([sq(x) for x in l if is_odd(x)])
```

With JavaScript a similar contrivance can also increase readability and facilitate DRY[27] code:

```
var isOdd = function(x){ return x%2; };

sum = l.filter(isOdd)
...
```

Map, reduce and filter with Python's lambdas

Although Python lacks anonymous functions it does have lambdas, nameless expressions which take arguments. While lacking the bells and whistles of JavaScript's anonymous functions these are a powerful addition to Python's functional programming repertoire, especially when combined with its functional methods.

> Python's functional built-ins, map, reduce, filter methods and lambda expressions, have a chequered past. It's no secret that the creator of Python wanted to remove them from the language. The clamour of disapproval lead to their reluctant preservation. With the recent trend towards functional programming this looks like a very good thing. They're not perfect but far better than nothing. And given JavaScript's strong functional emphasis they're a good way to leverage skills acquired in that language.

Python's lambdas take a number of parameters and return an operation on them, using a colon separator to define the function block, in much the same way as standard Python functions only pared to the bare essentials and with an implicit return. The following example shows a few lambdas employed in functional programming:

```
from functools import reduce # if using Python 3+

nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

odds = filter(lambda x: x % 2, nums)
odds_sq = map(lambda x: x * x, odds)
reduce(lambda x, y: x + y, odds_sq) ❶
Out:
165
```

27 Don't repeat yourself being a solid coding convention.

❶ Here the reduce method provides two arguments to the lambda, which uses them to return the expression after the colon.

JavaScript closures and the module-pattern

One of the key concepts in JavaScript is that of the closure, essentially a nested function declaration which uses variables declared in an outer (but not global) scope which are *kept alive* after the function is returned. Closures allow for a number of very useful programming patterns and are a common feature of the language.

Let's look at possibly the most common usage of closures and one we've already seen exploited in our module pattern (Example 2-2): exposing a limited API while having access to essentially private member variables.

A simple example of a closure is this little counter:

```
function Counter(inc) {
  var count = 0;
  var add = function() { ❶
    count += inc;
    console.log('Current count: ' + count);
  }
  return add;
}

var inc2 = Counter(2); ❷
inc2(); ❸
Out:
Current count: 2
inc2();
Out:
Current count: 4
```

❶ The `add` function gets access to the essentially private, outer-scope `count` and `inc` variables.

❷ This returns an add function with the closure-variables, count (0) and inc (2).

❸ Calling `inc2` calls `add`, updating the *closed* `count` variable.

We can extend the `Counter` to add a little API. This technique is the basis of JavaScript modules and many simple libraries. In essence it selectively exposes public method while hiding private method and

variables, generally seen as good practice in the programming world:

```
function Counter(inc) {
  var count = 0;
  var api = {};
  api.add = function() {
    count += inc;
    console.log('Current count: ' + count);
  }
  api.sub = function() {
    count -= inc;
    console.log('Current count: ' + count)
  }
  api.reset = function() {
    count = 0;
    console.log('Count reset to 0')
  }

  return api;
}

cntr = Counter(3);
cntr.add() // Current count: 3
cntr.add() // Current count: 6
cntr.sub() // Current count: 3
cntr.reset() // Count reset to 0
```

Closures have all sorts of uses in JavaScript and I'd recommend getting your head around them - you'll see them a lot as you start investigating other people's code. These are three particularly good web-articles that provide a lot of good use-cases for closures: 1, 2, 3.

Python has closures but they are not used nearly as much as JavaScript's, perhaps because of a few quirks which, though surmountable, make for some slightly awkward code. To demonstrate, Example 2-7 tries to replicate the previous JavaScript Counter.

Example 2-7. A first-pass attempt at a Python counter closure

```
def get_counter(inc):
    count = 0
    def add():
        count += inc
        print('Current count: ' + str(count))
    return add
```

If you create a counter with `get_counter` (Example 2-7) and try to run it you'll get an `UnboundLocalError`:

```
cntr = get_counter(2)
cntr()
Out:
...
UnboundLocalError: local variable 'count' referenced before
assignment
```

Interestingly, although we can read the value of count within the `add` function (comment out the `count += inc` line to try it), attempts to change it throw an error. This is because attempts to assign a value to something in Python assume it is local in scope. There is no `count` local to the `add` function and so an error is thrown.

In Python 3 we can get around the error in Example 2-7 by using the `nonlocal` keyword to tell Python that `count` is in a non-local scope:

```
...
def add():
    nonlocal count
    count += inc
...
```

In Python 2 we can use a little dictionary hack to allow mutation of our closed variables:

```
def get_counter(inc):
    vars = {'count': 0}
    def add():
        vars['count'] += inc
        print('Current count: ' + str(vars['count']))
    return add
```

This *hack* works because we are not assigning a new value to `vars` but mutating an existing container, perfectly valid even if it is out of local scope.

As you can see, with a bit of effort, JavaScripters can transfer their closure skills to Python. The use-cases are similar but Python, being a richer language with lots of useful batteries included, has more options to apply to the same problem. Probably the most common use of closures is in Python's decorators.

Decorators are essentially function wrappers that extend the functions' utility without having to alter the function itself. They're a relatively advanced concept but you can find a user-friendly introduction here.

This is that

One JavaScript hack you'll see a lot of is a consequence of closures and the slippery `this` keyword. If you wish to refer to the outer-scoped `this` in a child function then you must use a proxy as the child's `this` will be bound according to context. The convention is to use `that` to refer to `this`. The code is less confusing than the explanation:

```
function outer(bar){
   this.bar = bar;
   var that = this;
   function inner(baz){
       this.baz = baz * that.bar; ❶
       // ...
```

❶ `that` refers to the outer function's `this`

This concludes my cherry-picked selection of patterns and hacks I find myself using a lot in Dataviz work. You'll doubtless acquire your own but I hope these give you a leg up.

A Cheatsheet

As a handy reference guide, here's a set of cheat-sheets to translate basic operations between Python and JavaScript.

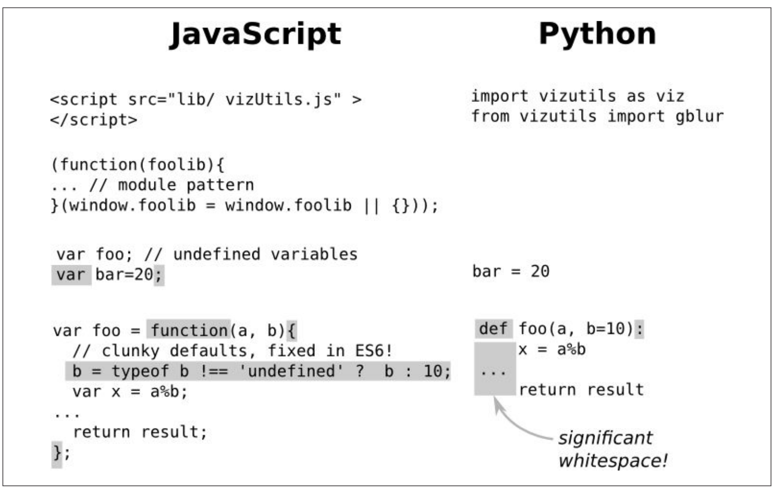


Figure 2-2. Some basic syntax

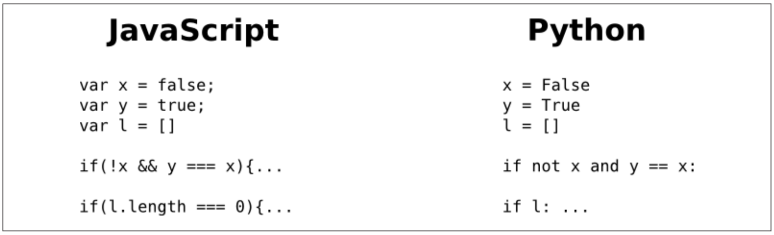


Figure 2-3. Booleans

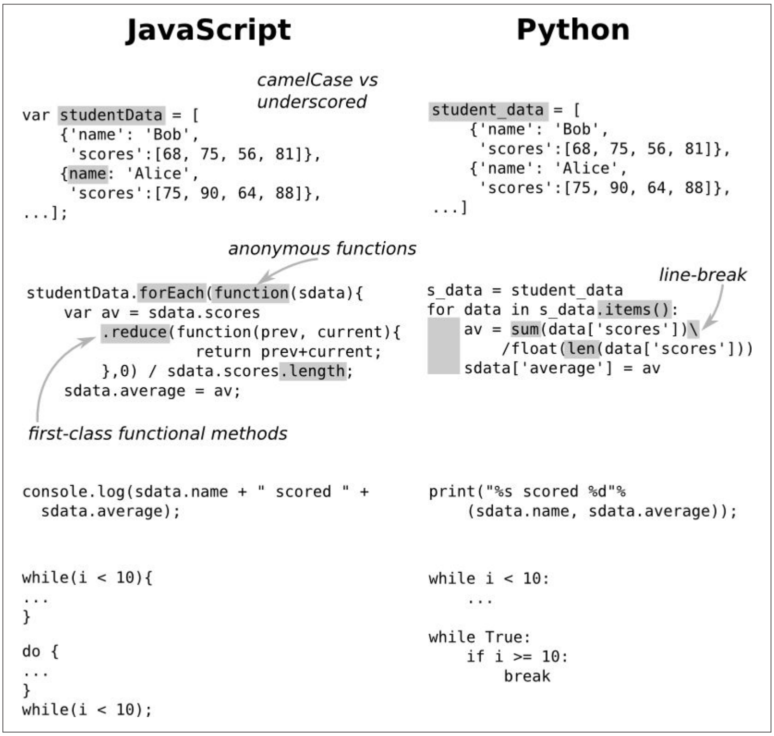


Figure 2-4. Loops and iterations

JavaScript

```
if(x === 'foo'){
    ...}
else if(x === 'bar'){
    ...}
else{
    ...}

if(x === foo && y !== bar){...

if(['foo', 'bar', 'baz']
     .indexOf(s) != -1){...

switch(foo){
  case bar:
    ...
    break;
  case baz: ...
  default:
    return false;
}
```

Python

```
if x == 'foo':
    ...
elif x == 'bar':
    ...
else:
    ...

if x == foo and y != bar:
    ...

if s in ['foo', 'bar', 'baz']:
    ...
```

Figure 2-5. Conditionals

JavaScript

```
var l = [1, 2, 3, 4];
l.push('foo'); // [...4, 'foo']
l.pop(); // 'foo', l=[..., 4]
l.slice(1,3) // [2, 3]
l.slice(-3, -1) // [2, 3]


l.map(function(o){ return o*o;})
// [1, 4, 9, 16]


d = {a:1, b:2, c:3};
d.a === d['a'] // 1
d.z // undefined

// OLD BROWSERS
for(key in d){
  if(d.hasOwnProperty(key){
    var item = d[key];

// NEW AND BETTER
Object.keys(d).forEach(key, i){
  var item = d[key];
```

Python

```
l = [1, 2, 3, 4]
l.append('foo') # [...4, 'foo']
l.pop() # 'foo', l=[..., 4]
l[1:3] # [2, 3]
l[-3:-1] # [2, 3]
l[0:4:2] # [1, 3] (stride of 2)

[o*o for o in l]
// [1, 4, 9, 16]


d = {'a':1, 'b':2, 'c':3}
d['a'] # 1
d.get('z') # NoneType
d['z'] # KeyError!

for key, value in d.items(): ...
for key in d:
for value in d.values():...
```

Figure 2-6. Containers

JavaScript

```javascript
var Foo = {
  initFoo: function(bar){
    this.bar = bar;
    return this;
  }
};

var Baz = Object.create(Foo);

Baz.initBaz = function(bar, qux){
  this.initFoo(bar);
  this.qux = qux;
  return this;
};

var baz = Object.create(Baz)
            .initBaz('answer', 42);
```

Python

```python
class Foo(object):
    def __init__(self, bar):
        self.bar = bar

class Baz(Foo):
    def __init__(self, bar, qux):
        super(Baz).__init__(bar)
        self.qux = qux

baz = Baz('answer', 42)
baz.bar # 'answer'
```

Figure 2-7. Classes and prototypes

Summary

I hope this chapter has shown that JavaScript and Python have a lot of common syntax and that most common idioms and patterns from one of the languages can be expressed in the other without too much fuss. The meat and potatoes of programming, iteration, conditionals, basic data manipulation etc. is simple in both languages and translation of functions straightforward. If you can program in one to any degree of competency, the threshold to entry for the other is low. That's the huge appeal of these simple scripting languages, which have a lot of common heritage.

I provided a list of patterns, hacks, idioms I find myself using a lot in dataviz work. I'm sure this list has its idiosyncrasies but I've tried to tick the obvious boxes.

Treat this as part-tutorial, part-reference for the chapters to come. Anything not covered here will be dealt with when introduced.

CHAPTER 3

Reading and Writing Data with Python

One of the fundamental skills of any data-visualiser is the ability to move data around. Whether your data is in an SQL database, a comma-separated-value (CSV) file or in some more esoteric form, you should be comfortable reading the data and being able to convert it and write it into a more convenient form if need be. One of Python's great strengths is how easy it makes manipulating data in this way and the focus of this chapter is to bring you up to speed with this essential aspect of our Dataviz toolchain.

This chapter is part tutorial, part reference and sections of it will be referred back in later chapters. If you know the fundamentals of reading and writing Python data you can cherry-pick parts of the chapter as a refresher.

Easy Does It

I remember when I started programming back in the day (using low-level languages like C) how awkward data manipulation was. Reading from and writing to files was an annoying mixture of boiler-plate code, hand-rolled kludges and the like. Reading from databases was equally difficult and as for serialising data, the memories are still painful. Discovering Python was a breath of fresh air. It wasn't a speed demon but opening a file was pretty much as simple as it could be:

```
file = open('data.txt')
```

Back then Python made reading from and writing to files refreshingly easy and its sophisticated string-processing made parsing the data in those files just as easy. It even had an amazing module called Pickle that could serialise pretty much any Python object.

In the years since, Python has added robust, mature modules to its standard library which make dealing with CSV and JSON files, the standard for web data-viz work, just as easy. There are also some great libraries for interacting with SQL-databases, such as SQLAlchemy, my thoroughly recommended go-to. The newer NoSQL-databases are also well served. MongoDB is by some way the most popular of these newer document-based databases and Python's `pymongo` library which, demonstrated later in the chapter, makes interacting with it a relative breeze.

Passing Data Around

A good way to demonstrate how to use the key data-storage libraries is to pass a single data packet among them, reading and writing it as we go. This will give us an opportunity to see in action the key data formats and databases employed by data-visualisers.

The data we'll be passing around is probably the most commonly used in web-visualisations, a list of dictionary-like objects (see Example 3-1). This data-set would be transferred to the browser in JSON form, which is, as we'll see, easily converted from a Python dictionary.

Example 3-1. Our target list of data objects

```
nobel_winners = [
 {'category': 'Physics',
  'name': 'Albert Einstein',
  'nationality': 'Swiss',
  'sex': 'male',
  'year': 1921},
 {'category': 'Physics',
  'name': 'Paul Dirac',
  'nationality': 'British',
  'sex': 'male',
  'year': 1933},
 {'category': 'Chemistry',
  'name': 'Marie Curie',
  'nationality': 'Polish',
  'sex': 'female',
```

```
  'year': 1911}
]
```

We'll start by creating a CSV file from the Python list shown in Example 3-1, as a demonstration of reading (opening) and writing system files.

The following sections assume you're in a work directory with a `data` sub-directory to hand. You can run the code from a Python interpreter or file.

Working with System Files

In this section we'll create a CSV-file from a Python list of dictionaries (Example 3-1). Usually you'd do this using the `csv` module, which we'll demonstrate after this section, so this is just a way of demonstrating basic Python file-manipulation.

First let's open a new file, using *w* as a second argument to indicate we'll be writing data to it.

```
f = open('data/nobel_winners.csv', 'w')
```

Now we'll create our CSV file from the nobel_winners dictionary (Example 3-1):

```
cols = nobel_winners[0].keys() ❶
cols.sort() ❷

with open('data/nobel_winners.csv, 'w') as f: ❸
    f.write(','.join(cols) + '\n') ❹

    for o in nobel_winners:
        row = [str(o[col]) for col in cols] ❺
        f.write(','.join(row) + '\n')
```

❶ Gets our data columns from the keys of the first object, i.e. "category, name, …".

❷ Sorts the columns in alphabetical order.

❸ Uses Python's `with` statement to guarantee the file is closed on leaving the block or if any exceptions occur.

❹ `join` creates a concatenated string from a list of strings (`cols` here), joined by the initial string, i.e. "category,name,..".

❺ Creates a list using the column keys to the objects in nobel_win ners.

Now we've created our CSV-file, lets use to Python to read it and make sure everything is correct:

```
with open('data/nobel_winners.csv') as f:
    for line in f.readlines():
        print(line),                    ❶

Out:
category,name,nationality,sex,year
Physics,Albert Einstein,Swiss,male,1921
Physics,Paul Dirac,British,male,1933
Chemistry,Marie Curie,Polish,female,1911
```

❶ Adding a , to the print statement inhibits the addition of an unnecessary new-line.

As the previous output shows, our CSV-file is well-formed. Let's use Python's built-in csv module to first read it and then create a CSV-file the right way.

CSV, TSV and Row-column Data-formats

Comma separated values (CSV) or their tab-separated cousins (TSV) are probably the most ubiquitous file-based data-formats and as a data-visualiser this will often be the forms you'll receive to work your magic with. Being able to read and write CSV files and their various quirky variants, such as pipe or semi-colon separated or those using ` in place of the standard double-quotes, is a fundamental skill and Python's csv module is capable of doing pretty much all your heavy-lifting here. Let's put it through its paces reading and writing our nobel_winners data:

```
nobel_winners = [
 {'category': 'Physics',
  'name': 'Albert Einstein',
  'nationality': 'Swiss',
  'sex': 'male',
  'year': 1921},
  ...
]
```

Writing our nobel_winners data (see Example 3-1) to a CSV file is a pretty simple affair. csv has a dedicated DictWriter class which will

turn our dictionaries into csv rows. The only piece of explicit book-keeping we have to do is write a header to our csv-file, using the keys of our dictionaries as fields (i.e. "category, name, nationality, sex'):

```
import csv
----
with open('data/nobel_winners.csv', 'wb') as f:
    fieldnames = nobel_winners[0].keys() ❶
    fieldnames.sort() ❷
    writer = csv.DictWriter(f, fieldnames=fieldnames)
    writer.writeheader() ❸
    for w in nobel_winners:
        writer.writerow(w)
----
<1> You need to explicitly tell the writer what fieldnames (in this case the 'cat
<2> We'll sort the CSV header-fields alphabetically for readability.
<3> Writes the CSV-file header ("category,name,...").
```

You'll probably be reading csv files more often than writing them[1]. Let's read back the `nobel_winners.csv` file we just wrote.

If you just want to use `csv` as a superior and eminently adaptable file line-reader, a couple of lines will produce a handy iterator, which can deliver your CSV rows as lists of strings:

```
import csv

with open('data/nobel_winners.csv') as f:
    reader = csv.reader(f)
    for row in reader: ❶
        print(row)

Out:
['category', 'name', 'nationality', 'sex', 'year']
['Physics', 'Albert Einstein', 'Swiss', 'male', '1921']
['Physics', 'Paul Dirac', 'British', 'male', '1933']
['Chemistry', 'Marie Curie', 'Polish', 'female', '1911']
```

❶ Iterates over the `reader` object, consuming the lines in the file.

Note that the numbers are read in string-form. If you want to manipulate them naturally you'll need to convert any numeric columns to their respective type, in this case integer years.

1 I recommend using JSON over CSV as your preferred data-format.

Usually a more convenient way to consume CSV data is to convert the rows into Python dictionaries. This *record* form is also the one we are using as our conversion target (a `list` of `dicts`). `csv` has a handy `DictReader` for just this purpose:

```
import csv

with open('data/nobel_winners.csv') as f:
    reader = csv.DictReader(f)
    nobel_winners = list(reader) ❶

nobel_winners

Out:
[{'category': 'Physics', 'nationality': 'Swiss', 'year': '1921',\
 'name': 'Albert Einstein', 'sex': 'male'},
{'category': 'Physics', 'nationality': 'British', 'year': '1933',\
 'name': 'Paul Dirac', 'sex': 'male'},
{'category': 'Chemistry', 'nationality': 'Polish', 'year': '1911',\
 'name': 'Marie Curie', 'sex': 'female'}]
```

❶ Inserts all of the `reader` items into a list.

As the output shows, we just need to cast the `dicts` year attributes to integers to conform `nobel_winners` to the chapter's target data (Example 3-1), thus:

```
for w in nobel_winners:
    w['year'] = int(w['year'])
```

The `csv` readers don't infer data-types from your file, interpreting everything as a string. Pandas, Python's pre-eminent data-hacking library, will try and guess the correct type of the data columns, usually successfully. We'll see this in action in the later dedicated Pandas chapters.

`csv` has a few useful arguments to help parse members of the CSV-family:

- *dialect*: by default *excel*, specifies a set of dialect-specific parameters. *excel-tab* is a sometimes used alternative.
- *delimiter*: usually files are comma-separated but they could use `|`, `:` or `' '` instead.
- *quotechar*: by default double-quotes are used but you occasionally find `|` or ``` ` ` ` ``` instead.

You can find the full set of csv parameters here.

Now we've successfully written and read our target data using the csv module, let's pass on our CSV-derived nobel_winners dict to the json module.

JSON

In this section we'll write and read our nobel_winners data using Python's json module. Let's remind ourselves of the data we're using;

```
nobel_winners = [
 {'category': 'Physics',
  'name': 'Albert Einstein',
  'nationality': 'Swiss',
  'sex': 'male',
  'year': 1921},
  ...
]
```

For data-primitives such as strings, integers, floats etc.., Python dictionaries are easily saved (or dumped in the JSON vernacular) into JSON-files, using the json module. The dump method takes a Python container and a file-pointer, saving the former to the latter:

```
import json

with open('data/nobel_winners.json', 'w') as f:
     json.dump(nobel_winners, f)

open('data/nobel_winners.json').read()

Out: '[{"category": "Physics", "name": "Albert Einstein",
"sex": "male", "person_data": {"date of birth": "14th March
1879", "date of death": "18th April 1955"}, "year": 1921,
"nationality": "Swiss"}, {"category": "Physics",
"nationality": "British", "year": 1933, "name": "Paul Dirac",
"sex": "male"}, {"category": "Chemistry", "nationality":
"Polish", "year": 1911, "name": "Marie Curie", "sex":
"female"}]'
```

Reading (or loading) a JSON-file is just as easy. We just pass the opened JSON-file to the json module's load method:

```
import json

with open('data/nobel_winners.json') as f:
    nobel_winners = json.load(f)

nobel_winners
```

```
Out:
[{u'category': u'Physics',
  u'name': u'Albert Einstein',
  u'nationality': u'Swiss',
  u'sex': u'male',
  u'year': 1921}, ❶
...
```

❶ Note that, unlike our CSV-file conversion, the integer type of the year column is preserved.

`json` has `loads` and `dumps` counterparts to the file access method, which load and dump JSON object-strings respectively.

Dealing with dates and times

Trying to dump a date(time) object to `json` produces a `TypeError`:

```
from datetime import datetime

json.dumps(datetime.now())
Out:
...
TypeError: datetime.datetime(2015, 9, 13, 10, 25, 52, 586792)
is not JSON serializable
```

When serializing simple data-types such as strings or numbers, the default `json` encoders and decoders are fine. But for more specialised data such as dates you will need to do your own encoding and decoding. This isn't as hard as it sounds and quickly becomes routine. Let's first look at encoding your Python `datetimes` into sensible JSON strings.

The easiest way to encode Python data containing `datetimes` is to create a custom encoder like the one shown in Example 3-2 which is provided to the `json.dumps` method as a *cls* argument. This encoder is applied to each object in your data in turn and converts and dates or date-times to their ISO-format string (see "Dealing with Dates, Times and Complex Data" on page 102).

Example 3-2. Encoding a Python `datetime` to JSON

```
import datetime
from dateutil import parser
import json
```

```
class JSONDateTimeEncoder(json.JSONEncoder): ❶
    def default(self, obj):
        if isinstance(obj, (datetime.date, datetime.datetime)): ❷
            return obj.isoformat()
        else:
            return json.JSONEncoder.default(self, obj)

def dumps(obj):
    return json.dumps(obj, cls=JSONDateTimeEncoder) ❸
```

❶ Subclasses a JSONEncoder in order to create customised date-handling one.

❷ Tests for a datetime object and if true returns the isoformat of any dates or datetimes, e.g. *2015-09-13T10:25:52.586792*

❸ Uses the *cls* argument to set our custom date-encoder.

Let's see how our new `dumps` method copes some `datetime` data:

```
now_str = dumps({'time': datetime.now()})
now_str
Out:
'{"time": "2015-09-13T10:25:52.586792"}'
```

The *time* field is correctly converted into an ISO-format string, ready to be decoded into a JavaScript `Date` object (see "Dealing with Dates, Times and Complex Data" on page 102 for a demonstration).

. While you could write a generic decoder to cope with date-strings in arbitrary JSON files[2], it's probably not advisable. Date-strings come in so many weird and wonderful varieties that this is a job best done by hand on what is pretty much always a known data-set.

The venerable `strptime` method, part of the `datetime.datetime` package is good for the job of turning a time-string in a known format into a Python `datetime` instance:

```
In [0]: time_str = '2012/01/01 12:32:11'

In [1]: dt = datetime.strptime(time_str, '%Y/%m/%d %H:%M:%S') ❶
```

2 The Python module `dateutil` has a parser that will parse most dates and times sensibly and might be a good basis for this.

```
In [2]: dt
Out[2]: datetime.datetime(2012, 1, 1, 12, 32, 11)
```

❶ `strptime` tries to match the time-string to a format string using various directives such as %Y (year with century) and %H (hour as a zero-padded decimal number). If successful it creates a Python `datetime` instance. See here for a full list of the directives available.

If `strptime` is fed a time-string that does not match its format it throws a handy `ValueError`:

```
dt = datetime.strptime('1/2/2012 12:32:11', '%Y/%m/%d %H:%M:%S')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-111-af657749a9fe> in <module>()
----> 1 dt = datetime.strptime('1/2/2012 12:32:11', '%Y/%m/%d %H:%M:%S')
...
ValueError: time data '1/2/2012 12:32:11' does not match
            format '%Y/%m/%d %H:%M:%S'
```

So to convert date fields of a known format into `datetimes` for a `data` list of dictionaries you could do something like this:

```
for d in data:
     try:
         d['date'] = datetime.strptime(d['date'], '%Y/%m/%d %H:%M:%S')
     except ValueError:
         print('Oops! - invalid date for ' + repr(d))
```

Now that we've dealt with the two most popular data file-formats let's shift to the big-guns and see how to read or data from and write our data to SQL and NoSQL databases.

SQL

For interacting with an SQL-database, `sqlalchemy` is by some way the most popular and, in my opinion, best Python library. It allows you to use raw SQL instructions if speed and efficiency is an issue but also provides a powerful object relational mapping (ORM) which allows you to operate on SQL-tables using a high-level, Pythonic API, treating them essentially as Python classes.

Reading and writing data using SQL while allowing the user to treat that data as a Python container is a complicated process and while `sqlalchemy` is far more user-friendly than using a low-level SQL-engine, it is still a fairly complex library. I'll be covering the basics

here, using our data as a target but would encourage you to spend a little time reading some of the rather excellent documentation here. Let's remind ourselves of the nobel_winners data-set we're aiming to write and read:

```
nobel_winners = [
 {'category': 'Physics',
  'name': 'Albert Einstein',
  'nationality': 'Swiss',
  'sex': 'male',
  'year': 1921},
  ...
]
```

Let's first write our target data to an SQLite file using SQLAlchemy, starting by creating the database engine.

Creating the database engine

The first thing you need to do when starting an `sqlalchemy` session is to create a database engine. This engine will establish a connection with the database in question and perform any conversions needed to the generic SQL instructions being generated by `sqlalchemy` and the data being returned.

There are engines for pretty much every popular database as well as a *memory* option, which holds the database in RAM, allowing fast access for testing etc.[3]. The great thing about these engines is that they are interchangeable, which means you could develop your code using the convenient file-based SQLite database and then switch in production to something a little more industrial, say Postgresql, by changing a single config-string. Check here for the full list of engines available.

The form for specifying a database URL is

```
dialect+driver://username:password@host:port/database
```

So, to connect to a *nobel_prize* MySQL database running on localhost would require something like this. Note that the

3 On a cautionary note, it is probably a bad idea to use different database configurations for testing and production.

`create_engine` does not actually make any SQL requests at this point, merely sets up the framework for doing so[4].

```
engine = create_engine('mysql://kyran:mypsswd@localhost/nobel_prize')
```

We'll use a file-based SQLite database, setting the *echo* argument to true, which will output any SQL instructions generated by SQLAlchemy. Note the use of three back-slashes after the colon:

```
from sqlalchemy import create_engine

engine = create_engine('sqlite:///data/nobel_prize.db', echo=True)
```

SQLAlchemy offers various ways to engage with databases but I would recommend using the more recent declarative style unless there are good reasons to go with something more low-level and fine-grained. In essence, with declarative mapping you sub-class your Python SQL-table classes from a base and SQLAlchemy introspects their structure and relationships. See here for more details.

Defining the database tables

We first create a `Base` class using `declarative_base`. This base will be used to create table-classes, from which SQLAlchemy will create the database's table schemas. You can use these table-classes to interact with the database in a fairly Pythonic fashion. Note that most SQL-libraries require you to formally define table-schemas. This is in contrast to such schemaless NoSQL variants as MongoDB. We'll take a look at the Dataset library later in this chapter, which enables schemaless SQL.

Using this `Base` we define our various tables, in our case a single `Winner` table. Example 3-3 shows how to subclass `Base` and use `sqlalchemy's` datatypes to define a table-schema. Note the __tablename__ member, which will be used to name the SQL-table and as a keyword to retrieve it, and the optional custom __repr__ method, which will be used when printing a table-row.

Example 3-3. Defining an SQL-database table

```
from sqlalchemy import Column, Integer, String, Enum
// ...
```

4 See details here of this *lazy initialization*.

```
class Winner(Base):
    __tablename__ = 'winners'

    id = Column(Integer, primary_key=True)
    name = Column(String)
    category = Column(String)
    year = Column(Integer)
    nationality = Column(String)
    sex = Column(Enum('male', 'female'))

    def __repr__(self):
      return "<Winner(name='%s', category='%s', year='%s')>"\
        %(self.name, self.category, self.year)
```

Having declared our Base subclasses in Example 3-3 we supply its metadata create_all method with our database engine to create our database[5]. Because we set the *echo* argument to true when creating the engine, we can see the SQL instructions generated by SQLAlchemy from the command-line:

```
Base.metadata.create_all(engine)

INFO:sqlalchemy.engine.base.Engine SELECT CAST('test plain
returns' AS VARCHAR(60)) AS anon_1
...
INFO sqlalchemy.engine.base.Engine
CREATE TABLE winners (
        id INTEGER NOT NULL,
        name VARCHAR,
        category VARCHAR,
        year INTEGER,
        nationality VARCHAR,
        sex VARCHAR(6),
        PRIMARY KEY (id),
        CHECK (sex IN ('male', 'female'))
)
...
INFO:sqlalchemy.engine.base.Engine:COMMIT
```

With our new winners table declared we can start adding winner instances to it.

5 This assumes the database doesn't already exist. If it does the Base will be used to create new insertions and to interpret retrievals.

Adding instances with a session

Now that we have created our database, we need a session to interact with:

```
from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind=engine)
session = Session()
```

We can now use our `Winner` class to create instances/database-rows and add them to the session:

```
albert = Winner(**nobel_winners[0]) ❶
session.add(albert)
session.new ❷
Out:
IdentitySet([<Winner(name='Albert Einstein', category='Physics',
             year='1921')>])
```

❶ Python's handy '**' operator unpacks our first `nobel_winners` member into key-value pairs, i.e. (name=*Albert Einstein*, category=*Physics*…).

❷ `new` is the set of any items that have been added to this session.

Note that all database insertions, deletions etc. take place in Python. It's only when we use the `commit` method that the database is altered.

.

> Use as few commits as possible, allowing SQLAlchemy to work its magic behind the scenes. When you commit, your various database manipulations should be summarised by SQLAlchemy and communicated in an efficient fashion. Commits involve establishing a database handshake and negotiating transactions, often a slow process and one you want to limit as much as possible, leveraging SQLAlchemy's book-keeping abilities to full advantage.

As the `new` method shows, we have added a Winner to the session. We can remove the object using `expunge`, leaving an empty `IdentitySet`:

```
session.expunge(albert) ❶
session.new
Out:
IdentitySet([])
```

❶ Remove the instance from the session (there is an `expunge_all` method which removes all new objects added to the session.).

At this point no database insertions or delections have taken place. Let's add all the members of our `nobel_winners` list to the session and commit them to the database:

```
winner_rows = [Winner(**w) for w in nobel_winners]
session.add_all(winner_rows)
session.commit()
Out:
INFO:sqlalchemy.engine.base.Engine:BEGIN (implicit)
...
INFO:sqlalchemy.engine.base.Engine:INSERT INTO winners (name,
category, year, nationality, sex) VALUES (?, ?, ?, ?, ?)
INFO:sqlalchemy.engine.base.Engine:(u'Albert Einstein',
u'Physics', 1921, u'Swiss', u'male')
...
INFO:sqlalchemy.engine.base.Engine:COMMIT
```

Now that we've committed our `nobel_winners` data to the database, let's see what we can do with it and how to recreate the target list Example 3-1.

Querying the database

To access data you use the `session's query` method, the result of which can be filtered, grouped, intersected etc., allowing the full range of standard SQL data retrieval. You can check out querying methods available here. For now I'll quickly run through some of the most common queries on our Nobel data-set.

Let's first count the number of rows in our winners' table:

```
session.query(Winner).count()
Out:
3
```

Next, let's retrieve all Swiss winners:

```
result = session.query(Winner).filter_by(nationality='Swiss') ❶
list(result)
Out:
[<Winner(name='Albert Einstein', category='Physics', year='1921')>]
```

❶ `filter_by` uses keyword-expressions, its SQL-expressions counterpart being `filter`, e.g. `filter(Winner.nationality == Swiss)`.

Now let's get all non-Swiss Physics winners:

```
result = session.query(Winner).filter(\
             Winner.category == 'Physics', Winner.nationality != 'Swiss')
list(result)
Out:
[<Winner(name='Paul Dirac', category='Physics', year='1933')>]
```

Here's how to get a row based on Id-number:

```
session.query(Winner).get(3)
Out:
<Winner(name='Marie Curie', category='Chemistry', year='1911')>
```

Now let's retrieve winners ordered by year:

```
res = session.query(Winner).order_by('year')
list(res)
Out:
[<Winner(name='Marie Curie', category='Chemistry', year='1911')>,
 <Winner(name='Albert Einstein', category='Physics', year='1921')>,
 <Winner(name='Paul Dirac', category='Physics', year='1933')>]
```

To reconstruct our target list requires a little effort converting the `Winner` objects returned by our session query into Python `dicts`. Let's write a little function to create a `dict` from an SQLAlchemy class. We'll use a little table-introspection to get the column labels (see Example 3-4).

Example 3-4. Converts a SQLAlchemy instance to a dict

```
def inst_to_dict(inst, delete_id=True):
    dat = {}
    for column in inst.__table__.columns: ❶
        dat[column.name] = getattr(inst, column.name)
    if delete_id:
        dat.pop('id') ❷
    return dat
```

❶ Accesses the instance's table class to get a list of column objects.

❷ If *delete_id* is true, remove the SQL primary id field.

We can use Example 3-4 to reconstruct our `nobel_winners` target list:

```
winner_rows = session.query(Winner)
nobel_winners = [inst_to_dict(w) for w in winner_rows]
nobel_winners
Out:
[{'category': u'Physics',
  'name': u'Albert Einstein',
  'nationality': u'Swiss',
  'sex': u'male',
  'year': 1921},
  ...
]
```

You can update database rows easily by changing the property of their reflected objects:

```
marie = session.query(Winner).get(3) ❶
marie.nationality = 'French'
session.dirty ❷
Out:
IdentitySet([<Winner(name='Marie Curie', category='Chemistry',
year='1911')>])
```

❶ Fetches Marie Curie, nationality Polish.

❷ `dirty` shows any changed instances not yet committed to the database.

Let's commit `marie's` changes and check that her nationality has changed from Polish to French:

```
session.commit()
Out:
INFO:sqlalchemy.engine.base.Engine:UPDATE winners SET
nationality=? WHERE winners.id = ?
INFO:sqlalchemy.engine.base.Engine:('French', 3)
...

session.dirty
Out:
IdentitySet([])

session.query(Winner).get(3).nationality
Out:
'French'
```

As well as updating database rows you can delete the results of a query:

```
session.query(Winner).filter_by(name='Albert Einstein').delete()
Out:
INFO:sqlalchemy.engine.base.Engine:DELETE FROM winners WHERE
winners.name = ?
INFO:sqlalchemy.engine.base.Engine:('Albert Einstein',)
1

list(session.query(Winner))
Out:
[<Winner(name='Paul Dirac', category='Physics', year='1933')>,
 <Winner(name='Marie Curie', category='Chemistry', year='1911')>]
```

You can also drop the whole table if required, using the declarative class's *table* attribute:

```
Winner.__table__.drop(engine)
```

In this section we've dealt with a single winners table, without any foreign-keys or relationship to any other tables, akin to a CSV or JSON file. SQLAlchemy adds the same level of convenience to dealing with such database tables many-to-one, one-to-many etc. relationships as it does to basic querying, using implicit joins, by supplying the `query` method with more than one table class, or explicitly using the query's `join` method. Check out the examples here for more details.

Easier SQL with Dataset

One library I've found myself using a fair deal recently is Dataset, a module designed to make working with SQL databases a little easier and more Pythonic than existing powerhouses like SQLAlchemy[6]. Dataset tries to provide the same degree of convenience you get when working with schemaless NoSQL databases such as MongoDB, removing a lot of the formal boilerplate, such as schema definitions, demanded by the more conventional libraries. Dataset is built on top of SQLAlchemy which means it works with pretty much all major databases and can exploit the power, robustness and maturity of that best-of-breed library. Let's see how it deals with reading and writing our target dataset (from Example 3-1).

Let's use the SQLite `nobel_prize.db` database we've just created to put Dataset through its paces:

6 Dataset's official motto being 'databases for lazy people'.

First we connect to our SQL database, using the same URL/file format as SQLAlchemy:

```
import dataset

db = dataset.connect('sqlite:///data/nobel_prize.db')
```

To get our list of winners we grab a table from our `db` database, using its name as a key, and then use the `find` method without arguments to return all winners:

```
wtable = db['winners']
winners = wtable.find()
winners = list(winners)
winners
Out:
[OrderedDict([(u'id', 1), (u'name', u'Albert Einstein'),
 (u'category', u'Physics'), (u'year', 1921), (u'nationality',
 u'Swiss'), (u'sex', u'male')]), OrderedDict([(u'id', 2),
 (u'name', u'Paul Dirac'), (u'category', u'Physics'),
 (u'year', 1933), (u'nationality', u'British'), (u'sex',
 u'male')]), OrderedDict([(u'id', 3), (u'name', u'Marie
 Curie'), (u'category', u'Chemistry'), (u'year', 1911),
 (u'nationality', u'Polish'), (u'sex', u'female')])]
```

Note that the instances returned by Dataset's `find` method are `OrderedDicts`. These useful containers are an extension of Python's `dict` class which behave just like dictionaries but remember the order in which items were inserted, meaning you can guarantee the result of iteration, pop the last item inserted etc. This is a very handy additional functionality.

.

> One of the most useful Python *batteries* for data-mungers is `collections`, from where `Dataset's` `OrderedDicts` come. The `defaultdict` and `Counter` classes are particularly useful. Check out what's available here.

Let's recreate our winners table with `Dataset`, first dropping the existing one:

```
wtable = db['winners']
wtable.drop()

wtable = db['winners']
wtable.find()
```

```
Out:
[]
```

To recreate our dropped winners table, we don't need to define a schema as with SQLAlchemy (see "Defining the database tables" on page 88). Dataset will infer that from the data we add, doing all the SQL creation implicitly. This is the kind of convenience one is used to when working with collection-based NoSQL databases. Let's use our `nobel_winners` dataset (Example 3-1) to insert some winner dictionaries. We use a database transaction and the `with` statement to efficiently insert our objects and then commit them[7].

```
with db as tx: ❶
    for w in nobel_winners:
        tx['winners'].insert(w)
```

❶ Use the `with` statement to guarantee the transaction `tx` is committed to the database.

Let's check that everything has gone well:

```
list(db['winners'].find())
Out:
[OrderedDict([(u'id', 1), (u'name', u'Albert Einstein'),
(u'category', u'Physics'), (u'year', 1921), (u'nationality',
u'Swiss'), (u'sex', u'male')]),
...
]
```

The winners have been correctly inserted and their order of insertion preserved by the `OrderedDict`.

Dataset is great for basic SQL-based work, particularly retrieving data you might wish to process or visualise. For more advanced manipulation it allows you to drop down into SQLAlchemy's core API using the `query` method.

On top of its huge convenience, Dataset has a `freeze` method which is a great asset to budding data-visualisers. `freeze` will take the result of an SQL-query and turn it into a JSON or CSV file, a very convenient way to start playing around with the data with JavaScript/D3:

```
winners = db['winners'].find()
dataset.freeze(winners, format='csv', \
```

7 See here for further details of how to use transactions to group updates.

```
                    filename='data/nobel_winners_ds.csv')

open('data/nobel_winners_ds.csv').read()
Out:
'id,name,category,year,nationality,sex\r\n
1,Albert Einstein,Physics,1921,Swiss,male\r\n
2,Paul Dirac,Physics,1933,British,male\r\n
3,Marie Curie,Chemistry,1911,Polish,female\r\n'
```

Now that we've covered the basics of working with SQL databases, let's see how Python makes working with the most popular NoSQL database just as painless.

MongoDB

Document-centric data-stores like MongoDB offer a lot of convenience to data wranglers. As with all tools, there are good and bad use-cases for NoSQL databases but if you have data that has already been refined and processed and don't anticipate needing SQL's powerful query language, based on optimised table-joins etc., MongoDB will probably prove easier to work with. MongoDB is a particularly good fit for web-dataviz because it uses Binary JSON (BSON) as its data-format. An extension of JSON, BSON can deal with binary-data, datetime-objects etc.. and plays very nicely with JavaScript.

Let's remind ourselves of the target data-set we're aiming to write and read:

```
nobel_winners = [
 {'category': 'Physics',
  'name': 'Albert Einstein',
  'nationality': 'Swiss',
  'sex': 'male',
  'year': 1921},
  ...
]
```

Creating a MongoDB collection with Python is the work of a few lines:

```
from pymongo import MongoClient

client = MongoClient() ❶
db = client.nobel_prize ❷
coll = db.winners ❸
```

❶ Creates a Mongo-client, using the default host and ports.

❷ Creates or accesses the *nobel_prize* database.

❸ If a *winners* collection exists this retrieves it, otherwise (as in our case) it creates it.

Using Constants for MongoDB Access

Accessing and creating a MongoDB database with Python involves the same operation, using dot-notation or square-bracket key-access:

```
db = client.nobel_prize
db = client['nobel_prize']
```

This is all very convenient but it means a single spelling mistake, e.g. *noble_prize*, could both create an unwanted database and future operations fail to update the correct one. For this reason I would advise using constant strings to access your MongoDB databases and collections:

```
DB_NOBEL_PRIZE = 'nobel_prize'
COLL_WINNERS = 'winners'

db = client[DB_NOBEL_PRIZE]
coll = db[COLL_WINNERS]
```

MongoDB databases run on localhost port 27017 by default but could be anywhere on the web. They also take an optional username and password. Example 3-5 shows how to create a simple utility function to access our database, with standard defaults.

Example 3-5. Accessing a MongoDB database

```
from pymongo import MongoClient

def get_mongo_database(db_name, host='localhost',\
                       port=27017, username=None, password=None):
    """ Get named database from MongoDB with/out authentication """
    # make Mongo connection with/out authentication
    if username and password:
        mongo_uri = 'mongodb://%s:%s@%s/%s'%\ ❶
        (username, password, host, db_name)
        conn = MongoClient(mongo_uri)
    else:
        conn = MongoClient(host, port)
```

```python
    return conn[db_name]
```

❶ We specify the database-name in the mongo URI (Uniform Resource Identifier) as the user may not have general privileges for the database.

We can now create a Nobel-prize database and add our target dataset (Example 3-1). Let's first get a winners collection, using the string constants for access:

```python
db = mongo_to_database(DB_NOBEL_PRIZE)
coll = db[COLL_WINNERS]
```

Inserting our Nobel dataset is then as easy as can be:

```python
coll.insert(nobel_winners)
Out:
[ObjectId('55f8326f26a7112e547879d4'),
 ObjectId('55f8326f26a7112e547879d5'),
 ObjectId('55f8326f26a7112e547879d6')]
```

The resulting array of `ObjectIds` can be used for future retrieval but MongoDB has already left its stamp on our `nobel_winners` list, adding a *hidden* id property[8]:

```python
nobel_winners
Out:
[{'_id': ObjectId('55f8326f26a7112e547879d4'),
  'category': u'Physics',
  'name': u'Albert Einstein',
  'nationality': u'Swiss',
  'sex': u'male',
  'year': 1921},
  ...
]
```

8 One of the cool things about MongoDB is that the `ObjectIds` are generated client-side, removing the need to quiz the database for them.

MongoDB's `ObjectIds` have quite a bit of hidden functionality, being a lot more than a simple random identifier. You can, for example, get the generation time of the ObjectId, giving you access to a handy time-stamp:

```
oid = bson.ObjectId()
oid.generation_time
Out: datetime.datetime(2015, 11, 4, 15, 43, 23...
```

Find the full details here.

Now that we've got some items in our *winners* collection, MongoDB makes finding them very easy its, with its `find` method taking a dictionary query:

```
res = coll.find({'category':'Chemistry'})
list(res)
Out:
[{u'_id': ObjectId('55f8326f26a7112e547879d6'),
  u'category': u'Chemistry',
  u'name': u'Marie Curie',
  u'nationality': u'Polish',
  u'sex': u'female',
  u'year': 1911}]
```

There are a number of special dollar-prefixed operators which allow for sophisticated querying. Let's find all the winners after 1930 using the *$gt* (greater-than) operator:

```
res = coll.find({'year': {'$gt': 1930}})
list(res)
Out:
[{u'_id': ObjectId('55f8326f26a7112e547879d5'),
  u'category': u'Physics',
  u'name': u'Paul Dirac',
  u'nationality': u'British',
  u'sex': u'male',
  u'year': 1933}]
```

You can also use Boolean expression, e.g. to find all winners after 1930 or female:

```
res = coll.find({'$or':[{'year': {'$gt': 1930}}, {'sex':'female'}]})
list(res)
Out:
[{u'_id': ObjectId('55f8326f26a7112e547879d5'),
  u'category': u'Physics',
  u'name': u'Paul Dirac',
  u'nationality': u'British',
  u'sex': u'male',
```

```
  u'year': 1933},
 {u'_id': ObjectId('55f8326f26a7112e547879d6'),
  u'category': u'Chemistry',
  u'name': u'Marie Curie',
  u'nationality': u'Polish',
  u'sex': u'female',
  u'year': 1911}]
```

You can find the full list of available query expressions here.

As a final test, let's turn our new *winners* collection back into a Python list of dictionaries. We'll create a little utility function for the task:

```
def mongo_coll_to_dicts(dbname='test', collname='test',\
                        query={}, del_id=True, **kw): ❶

    db = get_mongo_database(dbname, **kw)
    res = list(db[collname].find(query))

    if del_id:
        for r in res:
            r.pop('_id')

    return res
```

❶ An empty query dict {} will find all documents in the collection. `del_id` is a flag to remove MongoDB's `ObjectId's` from the items by default.

We can now create our target dataset:

```
mongo_coll_to_dicts(DB_NOBEL_PRIZE, COLL_WINNERS)
Out:
[{u'category': u'Physics',
  u'name': u'Albert Einstein',
  u'nationality': u'Swiss',
  u'sex': u'male',
  u'year': 1921},
  ...
]
```

MongoDB's schemaless databases are great for fast-prototyping in solo work or small teams. There will probably come a point, particularly with large code-bases, where some formal schema is a useful reference and sanity-check but while settling on the right data-model, the ease with which document forms can be adapted is a bonus.

Being able to pass Python dictionaries as queries to pymongo and having access to client-side generated ObjectIds are couple of other conveniences.

We've now passed the nobel_winners data in Example 3-1 through all our required file-formats and databases. Let's consider the special case of dealing with dates and times before summing up.

Dealing with Dates, Times and Complex Data

The ability to deal comfortably with dates and times is fundamental to data-viz work but can be quite tricky. There are many ways to represent a date or date-time as a string, each one requiring a separate encoding or decoding. For this reason it's good to settle on one format in your own work and encourage others to do the same. I'd recommend using the 1International Standard Organisation (ISO) time-format as your string representation for dates and times and using the Coordinated Universal Time (UTC) form[9]. Here's a few examples of ISO 8601 date and date-time strings:

2015-09-23	A date (Python/C format-code *%Y-%m-%d*)
2015-09-23T16:32:35Z	A UTC (*Z* after time) date and time (*T%H:%M:%S*)
2015-09-23T16:32+02:00	A positive two hour (+02:00) offset from UTC, e.g. Central European Time

Note the importance of being prepared to deal with different time-zones. These are not always on lines of longitude (see here) and often the best way to derive an accurate time is by UTC-time plus a geographic location.

ISO 8601 is the standard used by JavaScript and is easy to work with in Python. As web data-visualisers our key concern in creating a string representation that can be passed between Python and Javsacript using JSON and encoded and decoded easily at both ends.

9 To get the actual local time from UTC you can store a time-zone offset or, better still, derive it from a geo-coordinate; this is because time-zones do not follow lines of longitude very exactly

Let's take a date and time, in the shape of a Python `datetime`, convert it into a string, and then see how that string can be consumed by JavaScript.

First we produce our Python `datetime`:

```
from datetime import datetime

d = datetime.now()
d.isoformat()
Out:
'2015-09-15T21:48:50.746674'
```

This string can then be saved to JSON, CSV etc. read by JavaScript and used to create a `Date` object:

```
d = new Date('2015-09-15T21:48:50.746674')
> Tue Sep 15 2015 22:48:50 GMT+0100 (BST)
```

We can return the date-time to ISO 8601 string form with the `toISOString` method:

```
d.toISOString()
> "2015-09-15T21:48:50.746Z"
```

Finally, we can read the string back into Python.

If you know that you're dealing with ISO-format time-string, Python's `dateutil` module should do the job[10]]. But you'll probably want to sanity-check the result:

```
from dateutil import parser

d = parser.parse("2015-09-15T21:48:50.746Z")
d
Out:
datetime.datetime(2015, 9, 15, 21, 48, 50, 746000, tzinfo=tzutc())
```

Note that we've lost some resolution in the trip from Python to Javscript and back again, the latter dealing in milli not micro seconds. This is unlikely to be an issue in any dataviz work but is good to bear in mind, just in case some strange temporal errors occur.

10 To install just run 'pip install dateutil'. `dateutil` is a pretty powerful extension of Python's `datetime`, check it out here

Summary

This chapter aimed to make you comfortable using Python to move data around the various file-formats and databases a data-visualiser might expect to bump into. Using databases effectively and efficiently is a skill it takes a while to learn but you should now be comfortable with basic reading and writing for the large majority of dataviz use-cases.

Now we have the vital lubrication for our dataviz tool-chain, let's get up to scratch on the basic web-dev skills you'll need for the chapters ahead.

CHAPTER 4
Webdev 101

This chapter introduces the core web-development knowledge you will need to understand the web-pages you might want to scrape for data and to structure those you want to deliver, as the skeleton of your JavaScripted visualisations. As you'll see, in modern web-dev a little knowledge goes a long way, particularly when your focus is building self-contained visualisations and not whole web-sites (See "Single-page Apps" on page 106 for more details).

The usual caveats apply to a chapter in Part I — this chapter is part reference, part tutorial. There will probably be stuff here you know already so feel free to skip over it and get to the new material.

The Big Picture

The humble web-page, the collection of which compromises the World Wide Web (WWW) -that fraction of the internet consumed by humans- is constructed from files of various types. Apart from the multi-media files, images, videos, sound etc.., the key elements are textual, consisting of hypertext markup-language (HTML), cascading style sheets (CSS), and JavaScript. These three, along with any necessary data-files, are delivered using the Hypertext Transfer Protocol (HTTP) and used to build the page you see and interact with in your browser window, which is described by the Document Object Model (DOM), a hierarchical tree off which your content hangs. A basic understanding of how these elements interact is vital to building modern web visualisations and the aim of this chapter is to get you quickly up to speed.

Web-development is a big field and the aim here is not to turn you into a full-fledged web-developer. I assume you want to limit the amount of basic web-dev as much as possible, focusing only on that fraction necessary to build a modern visualisation. In order to build the sort of visualisations showcased at d3js.org , the New York Times or incorporated in basic interactive data-dashboards you actually need surprisingly little webdev fu. The result of your labours should be easily added to a larger web-site by someone dedicated to that job. In the case of small, personal web-sites it's trivial to do it yourself.

Single-page Apps

Single-page applications (SPAs) are web applications (or whole sites) which are dynamically assembled using JavaScript, often building from a lightweight HTML backbone and CSS-styles that can be applied dynamically with class and id attributes. Many modern data-visualisations fit this description, including the Nobel-prize visualisation that this book builds towards.

Often self-contained, the SPA's root-folder can be easily incorporated in an existing web-site or stand alone, REQUIRING only an HTTP server such as Apache or Nginx.

Thinking of our data-visualisations in terms of SPAs removes a lot of the cognitive overhead from the web-dev aspect of JavaScript visualisations, allowing us to focus on the programming challenges. The skills required to put the visualisation on the web are still fairly basic and quickly amortised. Often it will be someone else's job.

Tooling Up

As you'll see, the web-dev needed to make modern data-visualisations requires no more than a decent text-editor, modern browser and a terminal (Figure 4-1). I'll cover what I see as the minimal requirements for a web-dev ready editor and non-essential but nice-to-have features. My browser development tools of choice are Chrome's web-developer kit, freely available on all platforms. It has a lot of tab-delineated functionality and in this chapter I'll cover:

- The *Elements* tab, which allows you to explore the structure of a web-page, its HTML content, CSS styles and DOM presentation.

- The *Sources* tab, where most of your JavaScript debugging will take place.

You'll need a terminal for output, starting your local web-server, sketching ideas with the IPython interpreter etc..

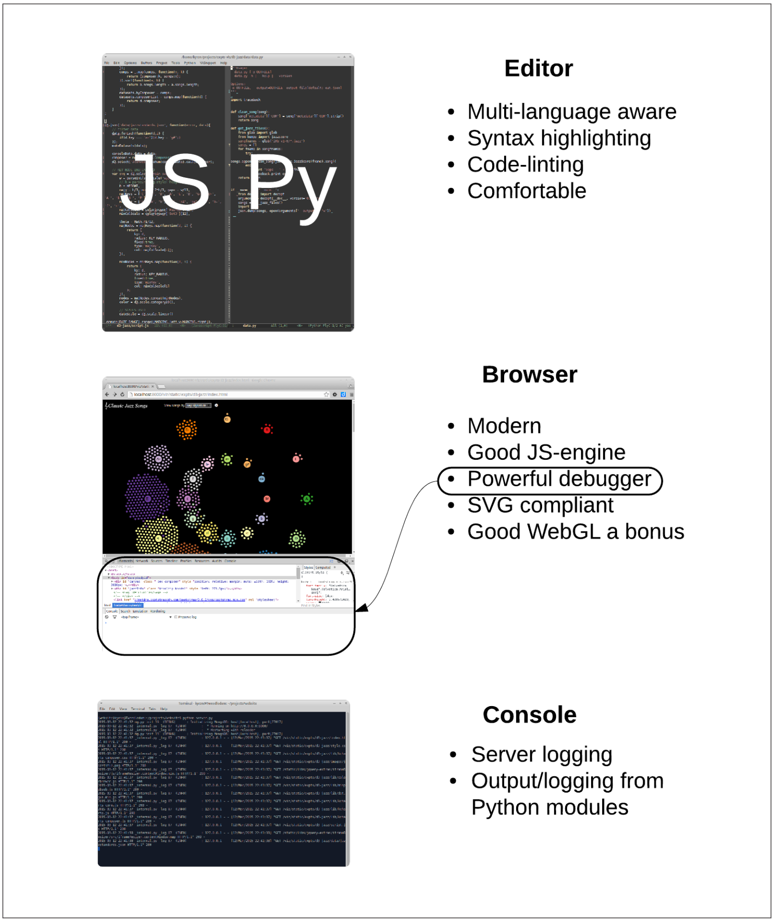


Figure 4-1. Primary Webdev Tools

Before dealing with what you do need, let's deal with a few things you really don't need when setting out, laying a couple of myths to rest on the way.

The myth of IDEs, frameworks and tools

There is a common assumption among the prospective JavaScripter that to program for the web requires a complex toolset, primarily an Intelligent Development Environment (IDE), as used by Enterprise - and other- coders everywhere. This is both potentially expensive and presents another learning curve. The good news is that not only have I never used an IDE to program for the web but I can't think of anyone I know in the discipline who does. In all probability the wonderful web-visualisations you have seen, that may have spurred you to pick up this book, were created with nothing more than a humble text-editor, a modern web-browser for viewing and debugging, and a console or terminal for logging and output.

There is also a commonly believed myth that one cannot be productive in JavaScript without using a framework of some kind. At the moment a number of these frameworks are vying for control of the JS ecosystem, sponsored by the various huge companies that created them. These frameworks come and go at a dizzying rate and my advice for anyone starting out in JavaScript is to ignore them entirely while you develop your core skills. Use small, targeted libraries, such as those in the jQuery ecosystem or Underscore's functional programming extensions and see how far you can get before needing a *my way or the highway* framework. Only lock yourself into a Framework to meet a clear and present need, not because the current JS group-think is raving about how great it is[1]. Another important consideration is that D3, the prime web datviz library, doesn't really play well with any of the bigger frameworks I know particularly the ones that want control over the DOM.

Another thing you'll find if you hang around web-dev forums, Reddit-lists, Stackoverflow etc.. is a huge range of tools constantly clamouring for attention. There are JS+CSS minifiers, watchers to automatically detect file-changes and reload web-pages during development etc. etc. While a few of these have their place, in my experience there are a lot of flaky tools which probably cost more time in hair-tearing than they gain in productivity. To reiterate, you can be very productive without such stuff and should only reach for one to scratch a current itch. Some, like Bower covered in this chap-

1 I bear the scars so you don't have to.

ter, are keepers but very few are remotely essential for data-visualisation work.

Your text editing work-horse

First and foremost among your webdev tools is a text-editor you are comfortable with and which can, at the very least, do syntax highlighting for multiple languages -in our case HTML, CSS, JavaScript and Python. You can get away with a plain, non-highlighting editor but in the long run it will prove a pain. Things like syntax-highlighting, code-linting, intelligent indentation and the like remove a huge cognitive load from the process of programming, so much so that I see their absence as a limiting factor. These are my minimal requirements for a text editor:

- Syntax highlighting for all languages you use.
- Configurable indentation levels and types for languages (e.g. Python 4 soft-tabs, JavaScript 2 soft-tabs).
- Multiple windows/panes/tabs to allow easy navigation around your code-base.

If you are using a relatively advanced text-editor, all the above should come as standard with the exception of code-linting which will probably require a bit of configuration.

My leading candidate for *nice to have* is a decent code-linter. If the mark of a useful tool is how much you would miss its absence then code-linting is easily in my top five. For scripting languages like Python and JavaScript, there's only so much intelligent code-analysis that can be achieved syntactically but just sanity-checking the obvious syntax errors can be a huge time save. In JavaScript in particular, some mistakes are transparent, in the sense that things will run in spite of them, and will quite often produce confusing error-messages. A code-linter can save you time here and enforce good practice. Figure 4-2 shows a contrived example of a JavaScript code-linter in action.

A recent addition to Ecmascript 5[2] is a *strict* mode, which enforces a modern JavaScript context. This mode is recognized by most linters

2 The specification for modern JavaScript is defined by the Ecmascript lineage.

and can be invoked by placing ‘use strict’ at the top of your program or within a function, to restrict it to that context. Modern browsers should also honour the strict mode, throwing errors for non-compliance. In strict mode trying to assign `foo = "bar";` will fail if `foo` hasn’t been previously defined. See John Resig`s nice explanation here.

```
 // you should use the function form of 'use strict'
!'use strict';
 // you included jQuery, but never used it
!(function ($) {
   // foo not defined
!  foo = 'baa';
   // pub is defined but never used
!  var pub = {
     // this is part of an object literal, not an assignment
!    init = function(response) {
       // respnse should be response
!      console.log(respnse);
     }
!  }
 // you're missing a semicolon here
 // also - its jQuery, not jquery
!}(jquery))
```

Figure 4-2. A running code-linter analyses the JavaScript continuously, highlighting syntax errors etc. in red and adding a ‘!’ to the left of the offending line.

Browser with development tools

One of the reasons an IDE is pretty much redundant in modern web-dev is that the best place to do debugging is in the web-browser itself and such is the pace of change there that any IDE attempting to emulate that context would have its work cut out. On top of this, modern web-browsers have evolved a powerful set of debugging and development tools. Firefox’s *Firebug* lead the way but has since been surpassed by *Chrome developer*, which offers a huge amount of functionality, from sophisticated (certainly to a Pythonista) debugging (parametric breakpoints, variable watches etc..) to memory and processor optimization profiling, device emulation (want to know what your web-page looks like on a smart-phone or tablet?) and a whole lot more. Chrome developer is my debugger of choice and will be used in this book. Like everything covered, it’s free as in beer.

Terminal or command-prompt

The terminal or command-line will be where you initiate the various servers and probably output the useful logging information. It's also where you'll try out Python modules etc.. or run a Python interpreter (*IPython* being by some way the best).

In OSX and Linux, this window is called a Terminal or xterm. In Windows it's a command-prompt which should be available through clicking Start→All-Programs→Accessories.

Building a Web-page

There are four elements to a typical web-visualisation:

- An HTML skeleton, with placeholders for our programmatic visualisation.
- Cascading Style Sheets (CSS) which define the look and feel (e.g. border widths, colors, font-sizes, placement of content-blocks).
- JavaScript to build the visualisation.
- Data to be transformed.

The first three of these are just text-files, created using our favourite editor and delivered to the browser by our web-server (of which more later - see ???). Let's examine them in turn.

Serving Pages with HTTP

The delivery of the HTML, CSS and JS files that are used to make a particular web-page (and any data-files, multimedia video, audio), is negotiated between a server and browser using the Hypertext Transfer Protocol. HTTP provides a number of methods, the most commonly used being *GET*, which requests a web resource, retrieving data from the server if all goes well or throwing an error if it doesn't. We'll be using *GET*, along with Python's requests module, to scrape some web-page content in Chapter 6.

To negotiate the HTTP browser generated HTTP requests you'll need a server. In development you can run a little server locally using Python's command-line initialised `SimpleHTTPServer`, like thus:

```
$ python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

This server is now serving content locally on port 8000. You can access the site it's serving by going to the URL *http://localhost:8000* on your browser.

`SimpleHTTPServer` is a nice thing to have and ok for demos and the like but is lacks a lot of basic functionality. For this reason, as we'll see in ???, it's better to master the use of a proper development (and production) server like Flask (this book's server of choice).

The DOM

The HTML files you send through HTTP are converted at the browser end into a Document Object Model or DOM, which can in turn be adapted by Javscript, this programmatic DOM being the basis of dataviz libraries like D3. The DOM is a tree structure, represented by hierarchical nodes, the top node being the main web-page or Document.

Essentially, the HTML you write or generate with a template is converted by the browser into a tree hierarchy of nodes, each one representing an HTML element. The top node is called the "Document Object" and all other nodes descend in a parent-child fashion. Programmatically manipulating the DOM is at the heart of such libraries as jQuery and the mighty D3 so it's vital to have a good mental model of what's going on. A great way to get the feel for the DOM is to use a web tool such as *Chrome Developer* (my recommended toolset) to inspect branches of the tree. ??? shows the DOM tree of a HTML page, accessible through the *Elements* tab.

Whatever you see rendered on the web-page, the book-keeping of the objects' state (displayed or hidden, matrix transform etc.) is being done with the DOM. D3's powerful innovation was to attach data directly to the DOM and use it to drive visual changes (Data Driven Documents).

The HTML skeleton

A typical web visualisation use an HTML skeleton, on which to build the visualisation with JavaScript.

HTML is the language used to describe the content of a web-page. It was first proposed by physicist Tim Berners Lee in 1980 while he

was working at the *CERN* particle accelerator complex in Switzerland. It uses tags such as `<div>`, `<image>`, `<h>` to structure the content of the page while CSS is used to define the look and feel[2]. The advent of HTML5 has reduced the boilerplate considerably but the essence is unchanged over those thirty years.

Fully specced HTML used to involve a lot of rather confusing header tags but with HTML5 some thought was put into a more user-friendly minimalism. This is pretty much the minimal requirement for a starting template[3]:

```
<!DOCTYPE html>
<meta charset="utf-8">
<body>
    <!-- page content -->
</body>
```

So we need only declare the document HTML, our character-set eight-bit unicode and a body tag below which to add our pages content. This is a big advance on the book-keeping required before and a very low threshold to entry, as far as creating the documents which will be turned into web-pages goes. Note the comment tag form: <!-- comment -->.

Now, more realistically we would probably want to add some CSS and JavaScript. You can add both directly to an HTML document by using <style> and <script> tags like this:

```
<!DOCTYPE html>
<meta charset="utf-8">
<style>
    <!-- CSS  -->
</style>
<body>
    <!-- page content -->
    <script>
        <!-- JavaScript -->
    </script>
</body>
```

This single-page HTML form is often used in examples such as those visualisations at d3js.org. It's convenient to have a single page

2 You can code style in HTML tags, using the `style` attribute, but it's generally bad practice. Better to use classes and ids defined in CSS.

3 as demonstrated by Mike Bostock here *http://bost.ocks.org/mike/d3/workshop/#8*, with a hat-tip to Paul Irish

to deal with when demonstrating code or keeping track of files but generally I'd suggest separating the HTML, CSS and JavaScript elements into separate files. The big win here, apart from easier navigation as the code-base gets larger, is that you can take full advantage of your editor's specific language enhancements such as solid syntax highlighting, code-linting (essentially syntax-checking on the fly) etc.. While some editors and libraries claim to deal with embedded CSS and JavaScript I haven't found an adequate one.

To use CSS and JavaScript files we just include them in the HTML using <link> and <script> tags like this:

```
<!DOCTYPE html>
<meta charset="utf-8">
<link rel="stylesheet" href="style.css" />
<body>
    <!-- page content -->
    <script type="text/javascript" src="script.js"></script> ❶
</body>
```

❶ Note the *async* directive to allow the browser to continue parsing the page while the script loads[2]

Marking-up content

Visualisations often use a small subset of the available HTML tags, usually building the page programmatically by attaching elements to the DOM-tree.

The most common tag is the <div>, marking a block of content. <div>s can contain other <div>s, allowing for a tree hierarchy, the branches of which are used during element selection and to propagate user-interface (UI) events such as mouse-clicks. Here's a simple `div` hierarchy:

```
<div id="my-chart-wrapper" class="chart-holder dev">
    <div id="my-chart" class="bar-chart">
         this is a placeholder, with parent #my-chart-wrapper
    </div>
</div>
```

Note the use of *id* and *class* attributes. These are used when selecting DOM elements and to apply CSS styles. id's are unique identifiers,

2 See this Stackoverflow thread for a good explanation *http://stackoverflow.com/questions/436411/where-is-the-best-place-to-put-script-tags-in-html-markup*.

each element should have only one and there should only be one occurrence of any particular id per page. The class can be applied to multiple elements, allowing bulk-selection, and each element can have multiple classes.

For textual content, the main tags are <p>, <h*> and
. You'll be using these a lot. This code produces Figure 4-3:

```
<h2>A Level-2 Header</h2>
<p>A paragraph of body-text with a line-break here..</br>
and a second paragraph...</p>
```

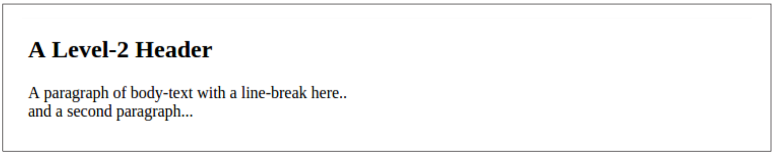


Figure 4-3. An h2 header and text

Header tags are reverse ordered by size from the largest <h1>.

`<div>`, `<h*>`, `<p>` are what is known as *block elements*. They normally begin and end with a new line. The other class of tag is *inline elements*, which display without line-breaks. Images <img>, hyperlinks <a>, and table cells <td> are among these, which include the <span> tag for inline text:

```
<div id="inline-examples">
    <img src="path/to/image.png" id="prettypic"> ❶
    <p>This is a <a href="link-url">link</a> to
        <span class="url">link-url</span></p> ❷
</div>
```

❶ Note that we don't need a closing tag for images.

❷ The span and link are continuous in the text

Other useful tags include lists, ordered <ol> and unordered <ul>:

```
<ol>
    <li>First item</li>
    <li>Second item</li>
</ol>
```

HTML also has a dedicated `<table>` tag, useful if you want to present raw data in your visualisation. This HTML produces the header and row in Figure 4-4:

```
<table id='chart-data'>
  <tr> ❶
    <th>Name</th>
    <th>Category</th>
    <th>Country</th>
  </tr>
  <tr> ❷
    <td>Albert Einstein</td>
    <td>Physics</td>
    <td>Switzerland</td>
  </tr>
</table>
```

❶ The header row

❷ The first row of data

Name	Category	Country
Albert Einstein	Physics	Switzerland

Figure 4-4. An HTML table

When making web visualisations, the most often used of the tags above are the textual tags, to provide instructions, information boxes etc.. But the meat of our JavaScripted efforts will probably be devoted to building DOM branches rooted on the Scalable Vector Graphics (SVG) <svg> and <canvas> tags. On most modern browsers the `<canvas>` tag also supports a 3D *WebGL* context, allowing *OpenGL* visualisations to be embedded in the page.

We'll deal with SVG, the focus of this book and the format used by the mighty D3 library, in a later section ("Scalable Vector Graphics (SVG)" on page 127). Now let's look at how we add style to our content blocks.

CSS

CSS, short for Cascading Style Sheets, is a language for describing the look and feel of a web-page. While you can hard-code style attributes into your HTML it's generally considered bad practice[2].

2 This is not the same as programmatically setting styles, which is a hugely powerful technique allowing styles to adapt to user interaction etc.

Much better to label your tag with an *id* or *class* and use that to apply styles in the stylesheet.

The key word in CSS is *cascading* - CSS follows a precedence rule so that in the case of a clash, the latest style overrides earlier ones. This means the order of inclusion for sheets is important. Usually you want your stylesheet to be loaded last so that you can override both the browser defaults and styles defined by any libraries you are using.

Figure 4-5 shows how CSS is used to apply styles to the HTML elements. First the element is selected, using hash #s to indicate a unique ID and dot .s to select members of a class. You then define one or more property→value pairs. Note that the font-family property can be a list of fallbacks, in order of preference. Here we want the browser default font-family of serif (capped strokes) to be replaced with the more modern sans-serif, with *Helvetica Neue* our first choice.

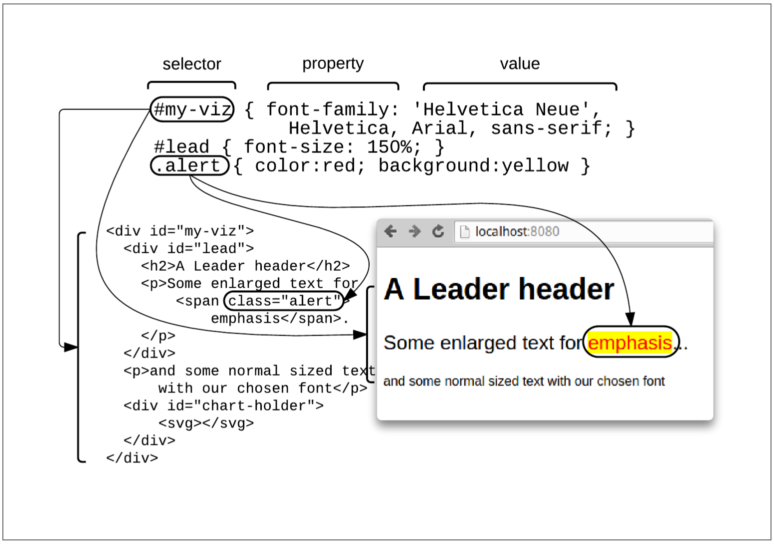


Figure 4-5. Styling the page with CSS

Understanding the CSS precedence rules is key to successfully applying styles. In a nutshell the order is

1. !important after CSS property trumps all.
2. The more specific the better. i.e. ids override classes.

3. The order of declaration - last declaration wins, subject to *1.* and *2.*

So, for example, say we have a <span> of class alert:

```
<span class="alert" id="special-alert">something to be alerted to</span>
```

Putting the following in our style.css file will make the alert text red and bold:

```
.alert { font-weight:bold; color:red }
```

If we then add this to the style.css, the id color black will override the class color red, while the class font-weight remains bold:

```
#special-alert {background: yellow; color:black}
```

To enforce the color red for alerts, we can use the !important directive:[2]

```
.alert { font-weight:bold; color:red !important }
```

If we then add another stylesheet, style2.css, after style.css:

```
<link rel="stylesheet" href="style.css" type="text/css" />
<link rel="stylesheet" href="style2.css" type="text/css" />
```

With style2.css containing the following:

```
.alert { font-weight:normal }
```

Then the font-weight of the alert will be reverted to normal, the new class style having been declared last.

JavaScript

JavaScript is the only first-class, browser-based programming language. In order to do anything remotely advanced (and that includes all modern web-visualizations) you should have a JavaScript grounding. Other languages which claim to make client-side/browser programming easier, such as Typescript, Coffeescript etc., compile to JavaScript, which means debugging either uses (generally flaky) mapping files or involves understanding the automated JavaScript. 99% of all web visualisation examples, the ones you should aim to be learning from, are in JavaScript and voguish alternatives have a way of fading with time. In essence, good competence in, if

2 This is generally considered bad practice and usually an indication of poorly structured CSS. Use with extreme caution as it can make life very difficult for co-developers.

not mastery of JavaScript is a pre-requisite for interesting web-visualisations.

The good news for Pythonistas reading is that JavaScript is actually quite a nice language, once you've tamed a few of its more awkward quirks[2]. As I showed in Chapter 2, JavaScript and Python have a lot in common and it's usually easy to translate from one to the other.

Data

The data needed to fuel you web visualisation will be provided by the web-server as static files (e.g. JSON or CSV files) or dynamically, through some kind of web-API (e.g. RESTful APIs) usually retrieving the data server-side from a database. We'll be covering all these forms in ???.

Although a lot of data used to be delivered in XML form, modern web-visualisation is predominantly about JSON and, to a lesser extent, CSV or TSV files.

JSON (short for Javascript Object Notation) is the de-facto web-visualisation data standard and I recommend you learn to love it. It obviously plays very nicely with Javascript but its structure will also be familiar to Pythonistas. As we saw in "JSON" on page 83, reading and writing JSON data with Python is a snip. Here's a little example of some JSON data:

```
{
  "firstName": "Groucho",
  "lastName": "Marx",
  "siblings": ["Harpo", "Chico", "Gummo", "Zeppo"],
  "nationality": "American",
  "yearOfBirth": 1890
}
```

Chrome's Developer Tools

The arms-race in JavaScript engines in recent years, which has produced huge increases in performance, has been matched by an increasingly sophisticated range of development tools built in to the various browers. Firefox's Firebug lead the pack but for a while now

2 These are succinctly discussed in Douglas Crockford's famously short *JavaScript the Good Parts*

Chrome's Developer Tools have surpassed it, adding functionality all the time. There's now a huge amount you can do with Chrome's tabbed tools but here I'll introduce the two most useful tabs, the HTML +CSS focused *Elements* and the Javscript focused *Sources*. Both of these work in complement to Chrome's developer console, demonstrated in "JavaScript" on page 36.

The Elements Tab

To access the Elements tab, select *More Tools* > *Developer Tools* from the right-hand options menu or use the Ctrl-Shift-I keyboard shortcut.

Figure 4-6 shows the elements tab at work. You can select DOM-elements on the page by using the left-hand magnifying glass and see their HTML-branch in the left-panel. The right panel allows you to see CSS styles applied to the element and look at any event-listeners attached or DOM-properties.

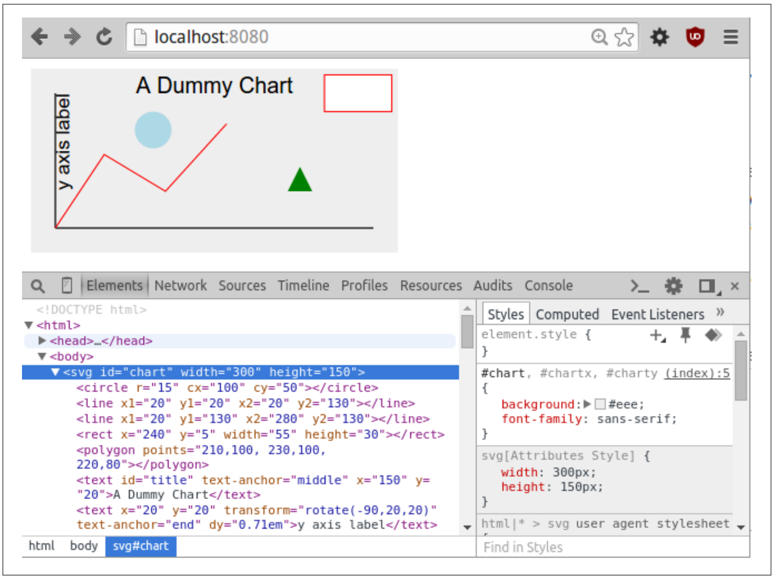


Figure 4-6. Chrome Developer Tools Elements tab

One really cool feature of the Elements tab is that you can interactively change element styling, both CSS styles and attributes[2]. This is a great way to refine the look and feel of your data visualisations.

Chrome's Elements tab provides a great way to explore the structure of a page, finding out how the different elements are positioned. This is good way to get your head around positioning content blocks with the *position* and *float* properties. Seeing how the pros apply CSS-styles is a really good way to up your game and learn some useful tricks.

The Sources Tab

The Sources tab allows you to see any JavaScript included in the page. Figure 4-7 shows the tab at work. In the left-panel you can select a script or an HTML-file with embedded `<script>` tagged JavaScript. As shown, you can place a breakpoint in the code, load the page and, on break, see the call-stack, any scoped or global variables etc.. These breakpoints are parametric so you can set conditions for them to trigger, handy if you want to catch and step through a particular configuration. On break you have the standard to step in, out and over functions etc..

2 Being able to play with attributes is particularly useful when trying to get Scalable Vector Graphics (SVG) to work.

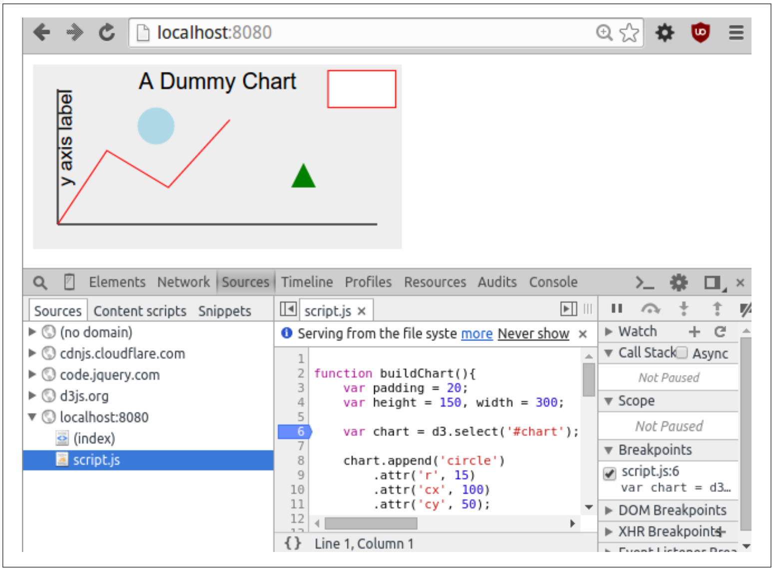


Figure 4-7. Chrome Developer Tools Sources tab

The Source tab is a fantastic resource and is the main reason why I hardly ever turn to console logging when trying to debug Javscript. In fact, where JS debugging was once a hit-and-miss black art, it is now almost a pleasure.

Other Tools

There's a huge amount of functionality in those Chrome Developer Tools tabs and it's being updated almost daily. You can do memory and CPU timelines and profiling, monitor your network downloads, test out your pages for different form-factors etc. But you'll spend 99% of your time as a data visualiser in the Elements and Sources tabs.

A Basic Page with Placeholders

Now that we have covered the major elements of a web-page, let's put them together. Most web-visualisations start off as HTML and CSS skeletons, with placeholder elements ready to be fleshed out with a little JavaScript plus data (see "Single-page Apps" on page 106).

We'll first need our HTML skeleton, using the code in Example 4-1. This consists of a tree of `<div>` content blocks, defining three chart-elements, a header, main and sidebar section.

Example 4-1. The HTML skeleton

```
<!DOCTYPE html>
<meta charset="utf-8">

<link rel="stylesheet" href="style.css" type="text/css" />

<body>

  <div id="chart-holder" class='dev'>
    <div id="header">
      <h2>A Catchy Title Coming Soon...</h2>
      <p>Some body-text describing what this visualisation is all
      about and why you should care.</p>
    </div>
    <div id="chart-components">
      <div id="main">
        A placeholder for the main chart..
      </div><div id="sidebar">
        <p>Some useful information about the chart,
          probably changing with user interaction...</p>
      </div>
    </div>
  </div>

  <script src="script.js"></script>
</body>
```

Now we have our HTML skeleton, we want to style it using some CSS. This will use the classes and ids of our content-blocks to adjust size, position, background color etc. To apply our CSS, in Example 4-1 we import a *style.css* file, shown in Example 4-2.

Example 4-2. CSS styling

```
body {
    background: #ccc;
    font-family: Sans-serif;
}

div.dev { ❶
    border: solid 1px red;
}
```

```
div.dev div {
    border: dashed 1px green;
}

div#chart-holder {
    width: 600px;
    background :white;
    margin: auto;
    font-size :16px;
}

div#chart-components {
    height :400px;
    position :relative; ❷
}

div#main, div#sidebar {
    position: absolute; ❸
}

div#main {
    width: 75%;
    height: 100%;
    background: #eee;
}

div#sidebar {
    right: 0; ❹
    width: 25%;
    height: 100%;
}
```

❶ This `dev` class is a handy way to see the border of any visual blocks, useful for visualisation work.

❷ Makes `chart-components` the relative parent.

❸ Makes the `main` and `sidebar` positions relative to `chart-components`.

❹ Positions this block flush with the right wall of `chart-components`.

We use absolute positioning of the main and siderbar chart elements (Example 4-2). There are various ways to position the content-blocks with CSS but absolute positioning gives you explicit control over their placement, a must if you want to get the look just right.

After specifying the size of the `chart-components` container, the `main` and `sidebar` child elements are sized and position using percentages of their parent. This means any changes to the size of `chart-components` will be reflected in its children.

With our HTML and CSS defined we can examine the skeleton by firing up Python's single-line `SimpleHTTPServer` in the project directory, like so:

```
$ python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

Figure 4-8 shows the resulting page with the Elements tab open, displaying the page's DOM-tree.

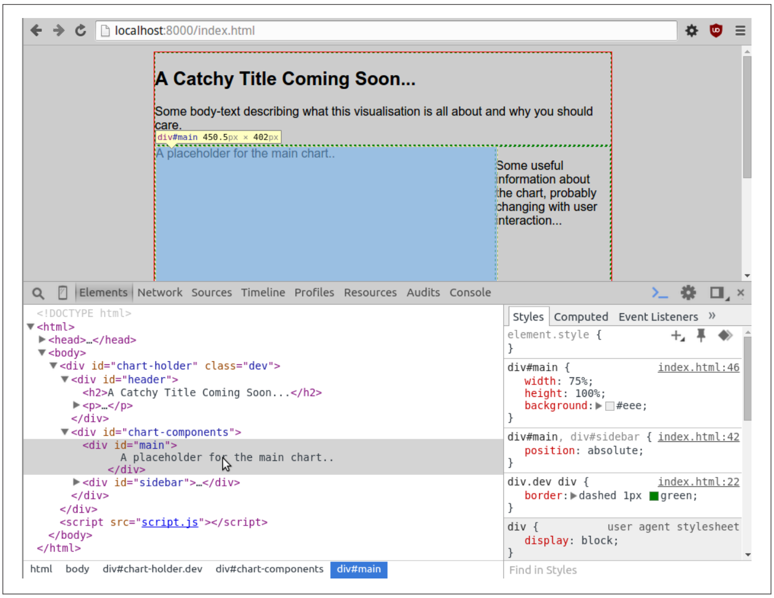


Figure 4-8. Building a Basic Webpage

The chart's content-blocks are now positioned and sized correctly, ready for JavaScript to add some engaging content.

Filling the placeholders with content

With our content blocks defined in HTML and positioned using CSS, a modern data visualisation uses JavaScript to construct its interactive charts, menus, tables and the like. There are many ways

to create visual content (aside from image or multimedia tags) on your modern browser, the main ones being:

- Scalable Vector Graphics (SVG) using special HTML-tags.
- Drawing to a 2D `canvas` context.
- Drawing to a 3D `canvas` WebGL context, allowing a subset of OpenGL commands.
- Using modern CSS to create animations, graphic primitives etc.

Because SVG is the language of choice for D3, by some way the biggest Javscript dataviz library, many of the cool web data-visualisations you have seen, such as those by the New York Times, are built in that. Broadly speaking, unless you anticipate having lots (>1000) of moving elements in your visualisation or need to use a specific `canvas` based library, SVG is probably the way to go.

By using vectors instead of pixels to express its primitives SVG will generally produce 'cleaner' graphics that respond smoothly to scaling operations. It's also much better at handling text, a crucial consideration for many visualisations. Another key advantage of SVG is that user interaction (e.g. mouse hovering or clicking) is native to the browser, being part of the standard DOM event handling[2]. A final point in its favour is that because the graphic components are built on the DOM, you can inspect and adapt them using your browser's development tools (see "Chrome's Developer Tools" on page 119). This can make debugging and refining your visualisations much easier than trying to find errors in the `canvas's` relatively black box.

`canvas` graphics contexts come into their own when you need to move beyond simple graphic primitives like circles and lines, for example incorporating images, such as pngs or jpegs. `canvas` is usually considerably more performant than SVG so anything with lots of moving elements[3] is better off rendered to a canvas. If you want to be really ambitious or move beyond 2D graphics, you can even unleash the awesome power of modern graphics cards by using a special form of canvas context, the OpenGL-based "webgl" context.

2 With a `canvas` graphic context you generally have to contrive your own event handling.

3 What this number is changes with time and the browser in question but as a rough rule of thumb the low thousands is where SVG often starts to strain.

Just bear in mind that what would be simple user interaction with SVG (e.g clicking on a visual element) often has to be derived from mouse-coordinates manually etc. adding a tricky layer of complexity.

The Nobel-prize data-visualisation realised at the end of this book's tool-chain is built primarily with D3 so SVG graphics are focus of this book. Being comfortable with SVG is fundamental to modern web-based dataviz so let's take a little primer.

Scalable Vector Graphics (SVG)

It doesn't seem long ago that Scalable Vector Graphics (SVG) seemed all washed up. Browser coverage was spotty and few big libraries were using it. It seemed inevitable that the `canvas` tag would act as a gateway to full-fledged, rendered graphics based on leveraging the awesome power of modern graphics cards. Pixels not vectors would be the building block of web-graphics and SVG would go down in history as a valiant but ultimately doomed 'nice idea'.

D3 might not single-handedly have rescued SVG in the browser but it must take the lion's share of responsibility. By demonstrating what can be done by using data to manipulate or drive the web-page's DOM it provided a compelling use-case for SVG. D3 really needs its graphic primitives to be part of the document hierarchy, in the same domain as the other HTML content. In this sense it needed SVG as much as SVG needed it.

The `svg` element

All SVG creations start with an `<svg>` root tag. All graphical elements such as circles, lines etc. and groups thereof are defined on this branch of the DOM-tree. Example 4-3 shows a little SVG context we'll use in upcoming demonstrations, a light-gray rectangle with id `chart`. We also include the D3 library, loaded from d3js.org and a `script.js` JavaScript file in the project folder.

Example 4-3. A basic SVG context

```
<!DOCTYPE html>
<meta charset="utf-8">
<!-- A few CSS style-rules -->
```

```
<style>
  svg#chart {
  background: lightgray;
  }
</style>

<svg id='chart' width="300" height="225">
</svg>

<!-- Third-party libraries and our JS script. -->
<script src="http://d3js.org/d3.v3.min.js"></script>
<script src="script.js"></script>
```

Now we've got our little SVG canvas in place, let's start doing some drawing.

The g element

We can group shapes within our `svg` element by using the group `g` element. As we'll see in "Working with groups" on page 137, shapes contained in a group can be manipulated together, e.g. changing their position, scale or opacity.

Circles

To create SVG visualisations, from the humblest little static bar-chart to full-fledged interactive, geographic masterpieces, involves putting together elements from a fairly small set of graphical primitives such as lines, circles and the very powerful paths. Each of these elements will have its own DOM-tag which will update as it changes[2]. e.g. e.g. it's *x* and *y* attributes will change to reflect any translations within its svg or group (g) context.

Let's add a circle to our SVG context to demonstrate:

```
<svg id='chart' width="300" height="225">
  <circle r="15" cx="100" cy="50"></circle>
</svg>
```

This produces Figure 4-9. Note that the y-coordinate is measured from the top of the `svg` *#chart* container, a common graphic convention.

2 You should be able to use your browser's development tools to see the tag attributes updating in real time.

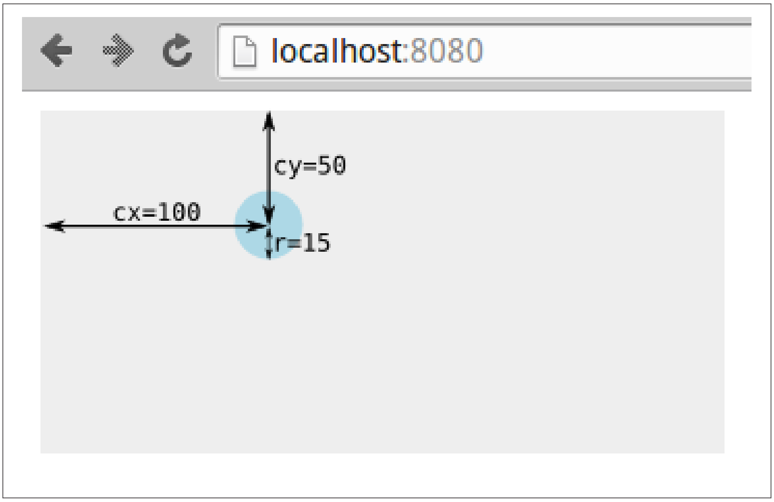


Figure 4-9. An SVG circle

Now let's see how we go about applying styles to SVG elements.

Applying CSS-styles

The circle in Figure 4-9 is fill-colored lightblue using CSS styling rules:

```
#chart circle{ fill: lightblue }
```

In modern browsers, most visual SVG styles can be set using CSS, including fill, stroke, stroke-width and opacity. So if we wanted a thick, semi-transparent green line (with id total) we could use the following CSS:

```
#chart line#total {
    stroke: green;
    stroke-width: 3px;
    opacity: 0.5;
}
```

You can also set the styles as attributes of the tags, though CSS is generally preferable.

```
<circle r="15" cx="100" cy="50" fill="lightblue"></circle>
```

Which SVG features can be set by CSS and which can't is a source of some confusion and plenty of gotchas. The SVG spec distinguishes between element properties and attributes, the former being more likely to be found among the valid CSS styles. You can investigate the valid CSS properties using Chrome's Elements tab and it's auto-complete. Also, be prepared for some surprises. For example, SVG text is colored using the *fill* not *color* property.

For fill and stroke, there are various color conventions you can use:

- named HTML colors, such as lightblue
- using HTML hex-codes (#RRGGBB), e.g. white #FFFFFF
- RGB values, e.g. red = rgb(255, 0, 0)
- RGBA values, where A is an alpha-channel (0-1), e.g half-transparent blue rgba(0, 0, 255, 0.5)

As well as adjusting the colors alpha-channel with RGBA, the SVG elements can be faded using their `opacity` property. Opacity is used a lot in D3 animations.

Stroke-width is measured in pixels by default but can use points etc..

Lines, rectangles, polygons

We'll add a few more elements to our chart to produce Figure 4-10.

First we'll add a couple of simple axis-lines to our chart, using the `<line>` tag. Line positions are defined by a start coordinate (x1, y1) and an end one (x2, y2):

```
<line x1="20" y1="20" x2="20" y2="130"></line>
<line x1="20" y1="130" x2="280" y2="130"></line>
```

We'll also add a dummy legend-box in the top-right corner using an SVG rectangle. Rectangles are defined by x and y coordinates relative to their parent container and a width and height:

```
<rect x="240" y="5" width="55" height="30"></rect>
```

You can create irregular polygons using the `<polygon>` tag, which takes a list of coordinate pairs. Let's make a triangle marker in the bottom right of our chart:

```
<polygon points="210,100, 230,100, 220,80"></polygon>
```

We'll style the elements with a little CSS:

```
#chart circle {fill: lightblue}
#chart line {stroke: #555555; stroke-width: 2}
#chart rect {stroke: red; fill: white}
#chart polygon {fill: green}
```

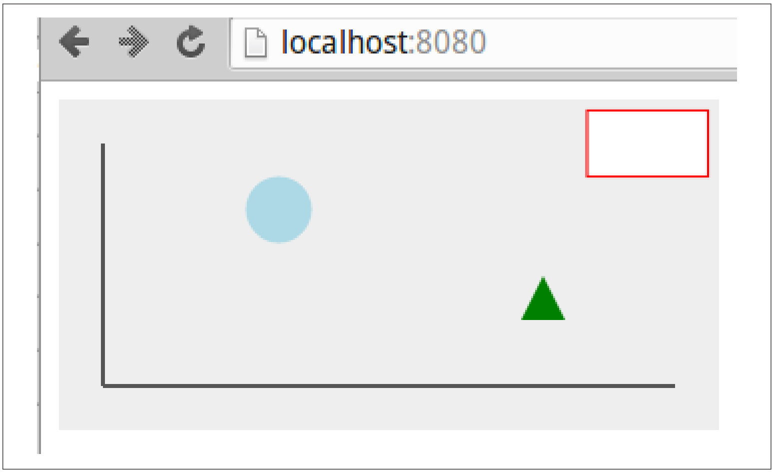


Figure 4-10. Adding a few elements to our dummy-chart

Now we've a few graphical primitives in place, let's see how we add some text to our dummy-chart.

Text

One of the key strengths of SVG over the rasertized `canvas` context is how it handles text. Vector-based text tends to look a lot clearer than its pixellated counterparts and benefits from smooth scaling too. You can also adjust stroke and fill properties, just like any SVG element.

Let's add a bit of text to our dummy-chart, a title and labelled y-axis (see Figure 4-11).

Text is placed using x and y coordinates. One important property is the `text-anchor` which stipulates where the text is placed relative to its x position. The options are *start*, *middle* and *end*, *start* being the default.

We can use the `text-anchor` property to center our chart title. We set the x coordinates at half the chart width and then set the `text-anchor` to *middle*:

```
<text id="title" text-anchor="middle" x="150" y="20">
  A Dummy Chart
</text>
```

Like all SVG primitives, we can apply scaling and rotation transforms to our text. To label our y-axis we'll need to rotate the text to the vertical (Example 4-4). By convention, rotations are clockwise by degree so we'll want an anti-clockwise, -90deg. rotation. By default rotations are about the (0,0) point of the element's container (`svg` or group `g`). We want to rotate our text about its own position so first translate the rotation point using the extra arguments to the `rotate` function. We also want to first set the `text-anchor` to the end of the 'y axis label' string to rotate about its end point.

Example 4-4. Rotating text

```
<text x="20" y="20" transform="rotate(-90,20,20)"
      text-anchor="end" dy="0.71em">y axis label</text>
```

In Example 4-4 we make use of the text's `dy` attribute which, along with `dx` can be used to make fine adjustments to the text's position. In this case we want to lower it so that when rotated anti-clockwise it will be to the right of the y-axis.

SVG text elements can also be styled using CSS. Here we set the font-family of the chart to `sans-serif` and the font-size to 16px, using the title id to make that a little bigger:

```
#chart {
background: #eee;
font-family: sans-serif;
}
#chart text{ font-size: 16px }
#chart text#title{ font-size: 18px }
```

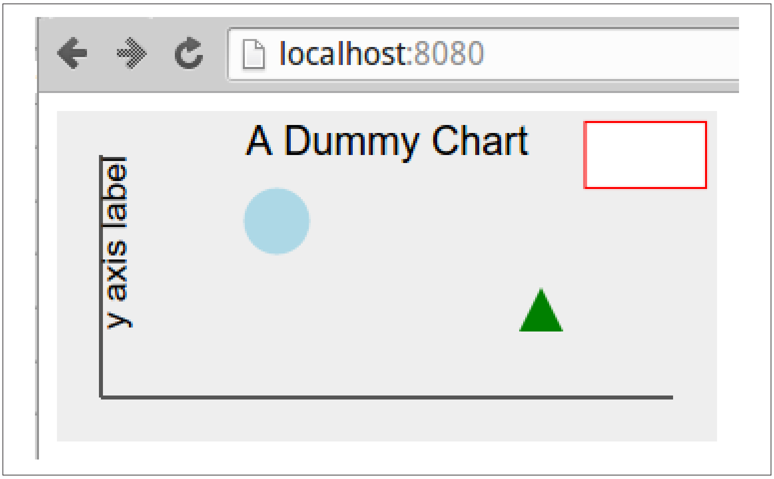


Figure 4-11. Some SVG text

Note that the `text` elements inherit font-family and font-size from the chart's CSS - you don't have to specify a `text` element.

Paths

Paths are the most complicated and powerful SVG element, enabling the creation of multi-line, multi-curve component paths which can be closed and filled, creating pretty much any shape you want. A simple example is adding a little chart-line to our dummy-chart to give Figure 4-12.

The red path in Figure 4-12 is produced by the following SVG:

```
<path d="M20,130L60,70L110,100L160,45"></path>
```

The `path's` `d` attribute specifies the series of operations needed to make the red-line. Let's break it down:

1. "M20, 130" - move to coordinate (20, 130)
2. "L60, 70" - draw a line to (60, 70)
3. "L110, 100" - draw a line to (110, 100)
4. "L160, 45" - draw a line to (160, 45)

You can imagine `d` as a set of instructions to a pen to move to a point with *M* raising the pen from the canvas.

A little CSS-styling is needed. Note that the `fill` is set to `none`; otherwise, to create a fill-area, the path would be closed, drawing a line from the its end to beginning points, and any enclosed areas filled in with the default color black:

```
#chart path {stroke: red; fill: none}
```

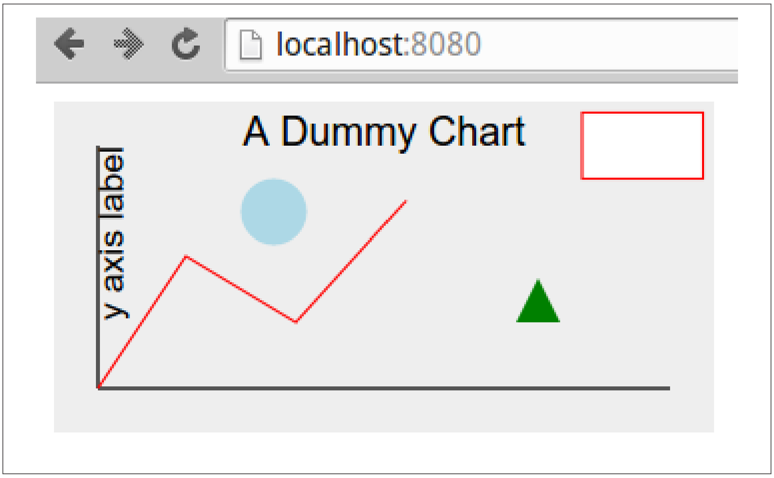


Figure 4-12. A red line-path from the chart axis

As well as the moveto 'M' and lineto 'L', the path has a number of other commands to draw arcs, Bezier curves etc.. SVG arcs and curves are commonly used in dataviz work, with many of D3's libraries making use of them[2]. Figure 4-13 shows some SVG elliptical-arcs created by the following code:

```
<svg id='chart' width="300" height="150">
  <path d="M40,40
           A30,40  ❶
           ,0,0,1, ❷
           80,80
           A50,50 ,0,0,1, 160, 80,
           A30,30 ,0,0,1, 190, 80
">
</svg>
```

❶ Having moved to position (40, 40), draw an elliptical-arc x-radius 30 and y-radius 40 and end-point (80, 80).

2 This chord-diagram is a nice example, using D3's `chord` function.

❷ The last two flags (0, 1) are `large-arc-flag`, specifying which arc of the ellipse to use and `sweep-flag` which specifies which of the two possible ellipses defined by start and end-points to use.



Figure 4-13. Some SVG Elliptical-arcs

The key flags used in the elliptical arc, `large-arc-flag` and `sweep-flag` are, like most things geometric, better demonstrated than described. Figure 4-14 shows the effect of changing the flags for the same relative beginning and end points, like so:

```
<svg id='chart' width="300" height="150">
  <path d="M40,80
           A30,40 ,0,0,1, 80,80
           A30,40 ,0,0,0, 120, 80,
           A30,40 ,0,1,0, 160, 80
           A30,40 ,0,1,1, 200, 80
">
</svg>
```



Figure 4-14. Changing the elliptic-arc flags

As well as lines and arcs, the path element offers a number of Bézier curves, quadratic, cubic and compounds of the two. With a little work these can realise any line-path you want. There's a nice run-through here with good illustrations.

For the definitive list of path elements and their arguments go here to the w3 source. And for a nice round-up see Jakob Jenkov's introduction

Scaling and rotating

As befits their vector nature, all SVG elements can be transformed by geometric operations. The most common used are *rotate*, *translate* and *scale* but you can also apply skewing using *skewX* and *skewY* or use the powerful, multi-purpose *matrix* transform.

Let's demonstrate the most popular transforms, using a set of identical rectangles. The transformed rectangles in Figure 4-15 are achieved like so:

```
<svg id='chart' width="300" height="150">
  <rect width="20" height="40" transform="translate(60, 55)"
        fill='blue'/>
  <rect width="20" height="40" transform="translate(120, 55),
        rotate(45)" fill='blue'/>
  <rect width="20" height="40" transform="translate(180, 55),
        scale(0.5)" fill='blue'/>
  <rect width="20" height="40" transform="translate(240, 55),
```

```
        rotate(45),scale(0.5)" fill='blue'/>
</svg>
```

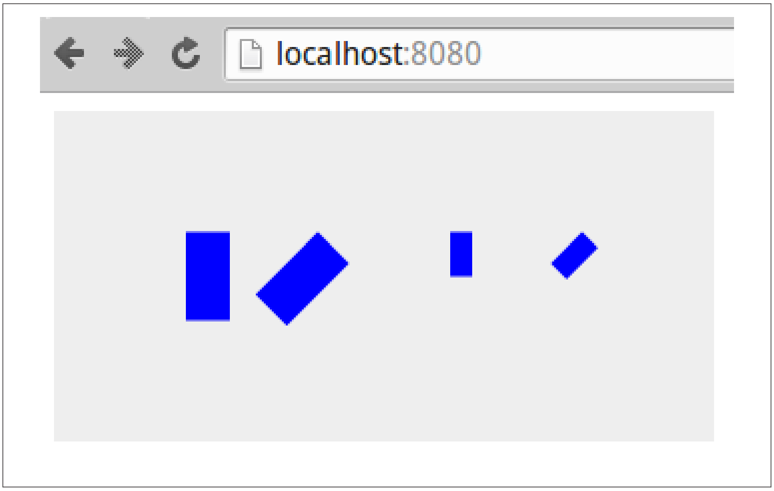
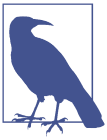


Figure 4-15. Some SVG transforms: rotate(45), scale(0.5), scale(0.5) + rotate(45)



The order in which transforms are applied is important. A rotation of 45 deg. clockwise foll-lowed by a translation along the x-axis will see the element moved south-easterly whereas the reverse operation moves it to the left and then rotates it.

Working with groups

Often when constructing a visualisation it's helpful to group the visual elements. A couple of particular uses being:

- When you require local coordinate schemes, e.g. if you have a text label for an icon you want to specify its position relative to the icon not the whole `svg` canvas.
- If you want to apply a scaling and/or rotation transformation to a sub-set of the visual elements.

SVG has a group <g> tag for this which you can think of as a mini canvas within the svg canvas. Groups can contain groups, allowing for very flexible geometric mappings[2].

Example 4-5 groups some shapes in the center of the canvas, producing Figure 4-16. Note that the position of circle, rect and path elements is relative to the translated group.

Example 4-5. Grouping some SVG shapes

```
<svg id='chart' width='300' height='150'>
  <g id='shapes' transform='translate(150,75)'>
    <circle cx='50' cy='0' r='25' fill='red' />
    <rect x='30' y='10' width='40' height='20' fill='blue' />
    <path d='M-20,-10L50,-10L10,60Z' fill='green' />
    <circle r='10' fill='yellow'>
  </g>
</svg>
```

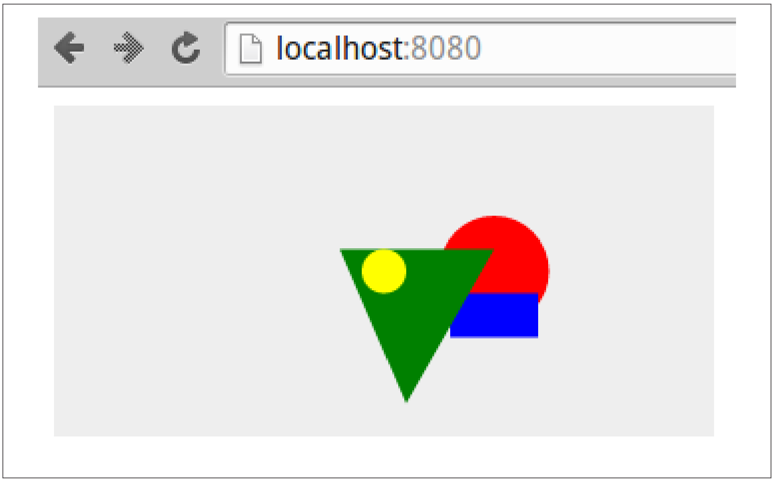


Figure 4-16. Grouping some shapes with SVG <g> tag

If we now apply a transform to the group, all shapes within it will be affected. Figure 4-17 shows the result of scaling Figure 4-16 by a factor of 0.75 and then rotating it 90, achieved by adapting the transform attribute, like so:

2 For example, a body group can contain an arm group can contain a hand group can contain finger elements.

```
<svg id='chart' width="300" height="150">
  <g id='shapes',
     transform = 'translate(150,75),scale(0.5),rotate(90)'>
     ...
</svg>
```

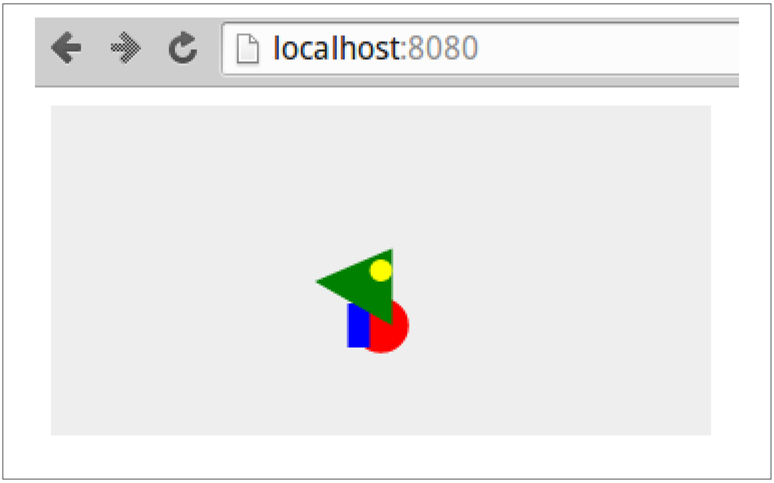


Figure 4-17. Transforming an SVG group

Layering and transparency

The order in which the SVG elements are added to the DOM-tree is important, with later elements taking precedence, layering over others. In Figure 4-16, for example, the triangle path obscures the red circle and blue rectangle and is in turn obscured by the yellow circle.

Manipulating the DOM ordering is an important part of JavaScripted dataviz, e.g. D3's `insert` method allows you to place an SVG element before an existing one.

Element transparency can be manipulated using the alpha-channel of `rgba(R,G,B,A)` colors or the more convenient `opacity` property. Both can be set using CSS. For overlaid elements, opacity is cumulative, as demonstrated by the color triangle Figure 4-18, produced by the following SVG:

```
<style>
  #chart circle { opacity: 0.33 }
</style>

<svg id='chart' width="300" height="150">
  <g transform='translate(150, 75)'>
```

```
    <circle cx='0' cy='-20' r='30' fill='red'/>
    <circle cx='17.3' cy='10' r='30' fill='green'/>
    <circle cx='-17.3' cy='10' r='30' fill='blue'/>
  </g>
</svg>
```

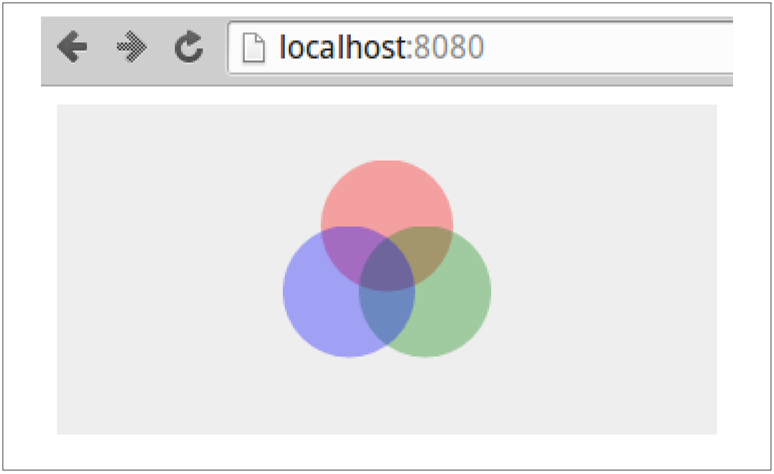


Figure 4-18. Manipulating opacity with SVG

The SVG elements demonstrated above were hand-coded in HTML but in data-visualisation work they are almost always added programmatically. Thus the basic D3 workflow is to add SVG elements to a visualisation, using data-files to specify their attributes and properties.

JavaScripted SVG

The fact that SVG graphics are described by DOM-tags has a number of advantages over a black-box such as the `<canvas>` context. For example, it allows non-programmers to create or adapt graphics and is a boon for debugging.

In web dataviz pretty much all your SVG elements will be created with JavaScript, using a library such as D3. You can inspect the results of this scripting using the browsers Element's tab ("Chrome's Developer Tools" on page 119), which is a great way to refine and debug your work, e.g. nailing an annoying visual glitch.

As a little taster for things to come, let's use D3 to scatter a few red circles on an SVG canvas. The dimensions of the canvas and circles are contained in a `data` object sent to a `chartCircles` function.

We use a little HTML place-holder for the SVG element:

```
<!DOCTYPE html>
<meta charset="utf-8">

<style>
  #chart circle {fill: red}
</style>

<body>
  <svg id='chart'>

  <script src="http://d3js.org/d3.v3.min.js"></script>
  <script src="script.js"></script>
</body>
```

With our place-holder SVG *chart* element in place, a little D3 in the `script.js` file is used to turn some data into the scattered circles (see Figure 4-19):

```
// script.js

var chartCircles = function(data) {

    var chart = d3.select('#chart');
    // Set the chart height and width from data
    chart.attr('height', data.height).attr('width', data.width);
    // Create some circles using the data
    chart.selectAll('circle').data(data.circles)
        .enter()
        .append('circle')
        .attr('cx', function(d) { return d.x })
        .attr('cy', function(d) { return d.y })
        .attr('r', function(d) { return d.r });
};

var data = {
    width: 300, height: 150,
    circles: [
        {'x': 50, 'y': 30, 'r': 20},
        {'x': 70, 'y': 80, 'r': 10},
        {'x': 160, 'y': 60, 'r': 10},
        {'x': 200, 'y': 100, 'r': 5},
    ]
};

chartCircles(data);
```



Figure 4-19. Some D3-generated circles

We'll see exactly how D3 works its magic in ???. For now, let's summarise what we've learned in this chapter.

Summary

This chapter provided a basic set of modern web-development skills for the budding data-visualiser. It showed how the various elements of a web-page (the HTML, CSS-stylesheets, JavaScript and media-files) are delivered by HTTP and, on being received by the browser, combined into the web-page the user sees. We saw how content blocks are described, using HTML tags such as `div` and `p`, and then styled and positioned using CSS. We also covered Chrome's Elements and Source tabs, the key browser development tools. Finally we had a little primer in SVG , the language in which most modern web data-visualisations are expressed. These skills will be extended when our toolchain reaches its D3 visualisation and new ones introduced in context.

PART II
Getting Your Data

In this part of the book we start our journey along the dataviz toolchain (see Figure II-1), beginning with a couple of chapters on how to get your data if it hasn't been provided for you.

In Chapter 5 we see how to get data off the web, using Python's `requests` library to grab web-based files and consume RESTful APIs. We also see how to use a couple of Python libraries that wrap more complex web-APIs, namely Twitter (with Python's Tweepy) and Google docs. The chapter ends with an example of light-weight web-scraping with the `BeautifulSoup` library.

In Chapter 6 we use Scrapy, Python's industrial-strength web-scraper, to get the Noble prize data-set we'll be using for our web-visualisation. With this *dirty* data-set to hand, we're ready for the next part of the book ???.

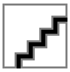


Figure II-1. Our dataviz toolchain: Getting the data

CHAPTER 5

Getting Data off the Web with Python

A fundamental part of the data-visualiser's skill-set is getting the right data-set, in as clean a form as possible. And more often than not these days, this involves getting it off the web. There are various ways you can do this and Python provides some great libraries which make sucking up the data easy.

The main ways to get data off the web are:

- Get a raw data-file over HTTP.
- Use a dedicated API to get the data.
- Scrape the data by getting web pages by HTTP and parsing them locally for content.

This chapter will deal with these ways in turn, but first let's get acquainted with the best Python HTTP library out there, `requests`.

Getting Web-data with the requests library

As we saw in Chapter 4, the files that are used by web-browsers to construct web-pages are communicated using the Hypertext Transfer Protocol, HTTP, first developed by Tim Berners Lee. Getting web-content, in order to parse it for data involves making HTTP requests.

Negotiating HTTP requests is a vital part of any general purpose language but getting web-pages with Python used to be a rather irksome affair. The venerable `urllib2` library was hardly user-friendly, with a very clunky API. `Requests`, courtesy of Kennith Reitz, changed that, making HTTP a relative breeze and fast establishing itself as the go-to Python HTTP library.

`Requests` is not part of the Python standard-library[2] but is part of the Anaconda package (see Chapter 1). If you're not using Anaconda, the following `pip` command should do the job:

```
$ pip install requests
Downloading/unpacking requests
...
Cleaning up...
```

If you're using a Python version prior to 2.7.9 then using `requests` may generate some Secure Sockets Layer (SSL) warnings. Upgrading to newer SSL libraries should fix this[3]:

```
$ pip intall --upgrade ndg-httpsclient
```

Now that you have `requests` installed, you're ready to perform the first task mentioned at the beginning of this chapter and grab some raw data-files off the web.

Getting Data-files with requests

A Python interpreter session is a good way to put `requests` through its paces so find a friendly local command-line, fire up IPython and import requests:

```
$ ipython
Python 2.7.5+ (default, Feb 27 2014, 19:37:08)
...

In [1]: import requests
```

To demonstrate, let's use the library to download a Wikipedia page. We use the `requests` library's `get` method to get the page and, by convention, assign the result to a `response` object.

2 This is actually a deliberate policy of the developers.

3 There are some platform dependencies that might still generate errors. This Stackoverflow thread is a good starting point if you still have problems.

```
response = requests.get("https://en.wikipedia.org/wiki/Nobel_Prize")
```

Let's use Python's `dir` method to get a list of the `response` object's attributes:

```
dir(response)
Out:
...
 'content',
 'cookies',
 'elapsed',
 'encoding',
 'headers',
 ...
 'iter_content',
 'iter_lines',
 'json',
 'links',
 ...
 'status_code',
 'text',
 'url']
```

Most of these attributes are self-explanatory and together provide a lot of information about the HTTP response generated. You'll use a small subset of these attributes generally. Firstly, let's check the status of the response:

```
response.status_code
Out: 200
```

As all good minimal web-devvers know, 200 is the HTTP status code for OK, indicating a successful transaction. Other than 200, the most common codes are:

- 401 (Unauthorized): attempting unauthorized access.
- 400 (Bad Request): trying to access the web-server incorrectly.
- 403 (Forbidden): similar to 401 but no login opportunity was available.
- 404 (Not Found): trying to access a web-page that doesn't exist.
- 500 (Internal Server Error): a general-purpose, catch-all error.

So, for example, if we made a spelling mistake with our request, asking to see the *SNoble_Prize* page, we'd get a 404 (Not Found) error:

```
response = requests.get("http://en.wikipedia.org/wiki/SNobel_Prize")
response.status_code
Out: 404
```

With our 200 OK response, from the correctly-spelled request, let's look at some of the info returned. A quick overview can be had with the `headers` property:

```
response.headers
Out: {
  'X-Client-IP': '104.238.169.128',
  'Content-Length': '65820', ...
  'Content-Encoding': 'gzip', ...
  'Last-Modified': 'Sun, 15 Nov 2015 17:14:09 GMT', ...
  'Date': 'Mon, 23 Nov 2015 21:33:52 GMT',
  'Content-Type': 'text/html; charset=UTF-8'...
  }
```

This shows, among other things, that the page returned was gzip encoded and 65k in size with content-type of text/html, encoded with unicode UTF-8.

Since we know text has been returned, we can use the `text` property of the response to see what it is:

```
response.text
Out: u'<!DOCTYPE html>\n<html lang="en"
dir="ltr" class="client-nojs">\n<head>\n<meta charset="UTF-8"
/>\n<title>Nobel Prize - Wikipedia, the free
encyclopedia</title>\n<script>document.documentElement.className =
```

This shows we do indeed have our Wikipedia HTML page, with some inline JavaScript. As we'll see in "Scraping Data" on page 160, in order to make sense of this content we'll need a parser to read the HTML and provide us with the content-blocks.

`requests` can be a convenient way of getting web-data into your program or Python session. For example, we can grab one of the datasets from the huge US government catalog, which often has the choice of various file-formats, e.g. JSON or CSV. Picking randomly, here's the data from a 2006-2010 study on food affordability, in JSON format. Note that we check that it has been fetched correctly, with a status_code 200:

```
response = requests.get(
"https://cdph.data.ca.gov/api/views/6tej-5zx7/rows.json\
?accessType=DOWNLOAD")

response.status_code
Out: 200
```

For JSON data, `requests` has a convenience method, allowing us to access the response data as a Python dictionary. This contains meta-data and a list of data-items:

```
data = response.json()
data.keys()
Out:
[u'meta', u'data']

data['meta']['view']['description']
Out: u'This table contains data on the average cost of a
market basket of nutritious food items relative to income for
female-headed households with children, for California, its
regions, counties, and cities/towns.  The ratio uses data from
the U.S. Department of Agriculture...

data['data'][0]
Out:
[1,
 u'4303993D-76F7-4A5C-914E-FDEA4EAB67BA',
 ...
 u'Food affordability for female-headed household with
 children under 18 years',
 u'2006-2010',
 u'1',
 u'AIAN',
 u'CA',
 u'06',
 u'California', ...
```

Now we've grabbed a raw page and a JSON file off the web, let's see how to use `requests` to consume a web data-API.

Using Python to Consume Data from a Web-API

If the data-file isn't on the web and you are lucky, rather than having to scrape some data configured for human consumption, there will be an Application Programming Interface (API) that enables you get the data programmatically and hopefully in a form that is cleaner and better organized (e.g. getting Twitter tweets from the official Twitter API).

The most popular data formats for web-APIs are JSON and XML, though a number of esoteric formats exist. For the purposes of the JavaScripting data-visualiser (discussed in ???), JavaScript Object

Notation (JSON) is obviously preferred. Helpfully, it is also starting to predominate.

There are different approaches to creating a web-API and for a few years there was a little war of the architectures. Three main types of API inhabit the web:

- REST: short for Representational state transfer, using a combination of HTTP verbs (GET, POST etc.) and Uniform Resource Identifiers (URIs), e.g. */user/kyran*, to access, create and adapt data.
- XML-RPC: a remote procedure call (RPC) protocol using XML encoding and HTTP transport.
- SOAP: short for Simple Object Access Protocol, using XML and HTTP.

This battle seems to be resolving in a victory for RESTful APIs and this is a very good thing. Quite apart from RESTful APIS being more elegant, easier to use and implement (see ???), some standardization here makes it much more likely that you will recognize and quickly adapt to a new API that comes your way. Ideally you will be able to reuse existing code.

Most access and manipulation of remote data can be summed up by the acronymn CRUD (create, retrieve, update, delete) originally coined to describe all the major functions implemented in relational databases. HTTP provides CRUD counterparts with the POST, GET, PUT and DELETE verbs and the REST abstraction builds on this use of these verbs, acting on a Universal Resource Identifier (URI).

Discussions about what is and isn't a proper RESTful interface can get quite involved[2] but essentially the URI (e.g. *http://example.com/api/items/2)* should contain all the information required in order to perform a CRUD operation. The particular operation (e.g. GET or DELETE) is specified by the HTTP verb. This excludes architectures such as SOAP which place stateful information in meta-data on the requests header. Imagine the URI as the virtual address of the data and CRUD all the operations you can perform on it.

2 See Parkinson's law of triviality, also known as bike-shedding.

As data visualizers, keen to lay our hands on some interesting datasets, we are avid consumers here, so our HTTP verb of choice is GET and the examples below will focus on the fetching of data with various well known web-APIs. Hopefully some patterns will emerge.

Although the two constraints of stateless URIs and the use of the CRUD verbs is a nice constraint on the shape of RESTless APIs, there still manage to be many varients on the theme.

Using a RESTful Web-API with requests

`requests` has a fair number of bells and whistles based around the main HTTP request verbs. For a good overview see here. For the purposes of getting data, you'll use GET and POST pretty much exclusively with GET being by a long way the most used verb. POST allows you to emulate web-forms, including login details, field-values etc. in the request. For those occasions where you find yourself driving a web-form with, for example, lots of options selectors, requests makes automation with POST easy. GET covers pretty much everything else, including the ubiquitous RESTful APIs, which provide an increasing amount of the well-formed data available on the web.

Let's look at a more complicated use of `requests`, getting a URL with arguments. The Organisation for Economic Cooperation and Development (OECD) provides some useful datasets on its site. The API is described here and queries are constructed using the dataset name (dsname), some dot-separated dimensions, each of which can be a number of + separated values. The URL can also take standard HTTP parameters initiated by a `?` and separated by `&`s:

```
<root_url>/<dsname>/<dim 1>.<dim 2>...<dim n>/all?param1=foo&param2=baa..
<dim 1> = 'AUS'+'AUT'+'BEL'...
```

So the following is a valid URL:

```
http://stats.oecd.org/sdmx-json/data/QNA      ❶
    /AUS+AUT.GDP+B1_GE.CUR+VOBARSA.Q          ❷
    /all?startTime=2009-Q2&endTime=2011-Q4    ❸
```

❶ Specifies the QNA dataset.

❷ Four dimensions, by location, subject, measure and frequency.

❸ Data from the second quarter 2009 to fourth quarter 2011.

Let's construct a little Python function to query the OECD's API:

Example 5-1. Making a URL for the OECD API

```
OECD_ROOT_URL = 'http://stats.oecd.org/sdmx-json/data'

def make_OECD_request(dsname, dimensions, params=None, \
root_dir=OECD_ROOT_URL):
    """ Make a URL for the OECD-API and return a response """

    if not params: ❶
        params = {}

    dim_str = '.'.join('+'.join(d) for d in dimensions) ❷
    url = root_dir + '/' + dsname + '/' + dim_str + '/all'
    print('Requesting URL: ' + url)
    return requests.get(url, params=params) ❸
```

❶ You shouldn't use mutable values, such as {}, for Python function defaults. See here for an explanation of this gotcha.

❷ Using a list comprehension with Python's succinct string concatenator `join`.

❸ Note that `request's get` can take a parameter dictionary as its second argument, using it to make the URL-query string.

We can use this function like so, to grab economic data for the USA and Australia from 2009-2010:

```
response = make_OECD_request('QNA',
    (('USA', 'AUS'),('GDP', 'B1_GE'),('CUR', 'VOBARSA'), ('Q')),
    {'startTime':'2009-Q1', 'endTime':'2010-Q1'})
```

```
Requesting URL: http://stats.oecd.org/sdmx-json/data/QNA/
    USA+AUS.GDP+B1_GE.CUR+VOBARSA.Q/all
```

Now to look at the data, we just check the response is OK and have a look at the dictionary keys:

```
if response.status_code == 200:
   json = response.json()
   json.keys()
Out: [u'header', u'dataSets', u'structure']
```

The resulting JSON data is in the SDMX format, designed to facilitate the communication of statistical data. It's not the most intuitive format around but it's often the case that data-sets have a less than

ideal structure. The good news is that Python is a great language for knocking data into shape. For Python's Pandas library (see ???) there is pandaSDMX which currently handles the XML based format.

The OECD API is essentially RESTful[2] with all the query being contained in the URL and the HTTP verb GET specifying a fetch operation. If a specialised Python library isn't available to use the API, e.g. Tweepy for Twitter, then you'll probably end up writing something like Example 5-1. `requests` is a very friendly, well designed library and can cope with pretty much all the manipulations required to use a web-API.

Getting some country data for the Nobel-viz

There are some national statistics which will come in handy for the Nobel-visualisation we're using our tool-chain to build. Population sizes, three-letter international codes (e.g. GDR, USA), geographic centers etc., are potentially useful when visualising an international prize and its distribution. REST-countries is a handy RESTful web-resource with various international stats. Let's use it to grab some data.

Requests to REST-countries take the following form:

```
https://restcountries.eu/rest/v1/<field>/<name>?<params>
```

As with the OECD API (see Example 5-1), we can make a simple calling function to allow easy access to the API's data, like so:

```
def REST_country_request(field, name, params=None):

    headers = {'User-Agent': 'Mozilla/5.0'} ❶

    if not params:
        params = {}

    if field == 'all':
        return requests.get(REST_EU_ROOT_URL + '/all', headers=headers)

    url = '%s/%s/%s'%(REST_EU_ROOT_URL, field, name)
    print('Requesting URL: ' + url)
    return requests.get(url, params=params, headers=headers)
```

<1>

2 SDMX is a RESTful specification.

With the Request_country_EU function to hand, let's get a list of all the countries using the US-dollar as currency:

```
response = REST_country_request('currency', 'usd')
if response.status_code == 200: # request OK
    response.json()
Out:
[{u'alpha2Code': u'AS',
  u'alpha3Code': u'ASM',
  u'altSpellings': [u'AS',
  ...
  u'capital': u'Pago Pago',
  u'currencies': [u'USD'],
  u'demonym': u'American Samoan',
  ...
  u'latlng': [12.15, -68.266667],
  u'name': u'Bonaire',
  ...
  u'name': u'British Indian Ocean Territory',
  ...
  u'name': u'United States Minor Outlying Islands',
  ...
```

The full data-set at REST-countries is pretty small so for convenience we'll make a copy and store it locally to MongoDB and our *nobel-prize* database using the get_mongo_database method from "MongoDB" on page 97:

```
db_nobel = get_mongo_database('nobel_prize')
col = db_nobel['country_data'] # country-data collection

# Get all the RESTful country-data
response = REST_country_request()
if response.status_code == 200:
    # Insert the JSON-objects straight to our collection
    col.insert(country_data)
Out:
[ObjectId('5665a1ef26a7110b79e88d49'),
 ObjectId('5665a1ef26a7110b79e88d4a'),
 ...
```

With our country-data inserted to its MongoDB collection, let's again find all the countries using the US-dollar as currency:

```
res = col.find({'currencies':{'$in':['USD']}})
list(res)
Out:
[{u'_id': ObjectId('5665a1ef26a7110b79e88d4d'),
  u'alpha2Code': u'AS',
  u'alpha3Code': u'ASM',
  u'altSpellings': [u'AS',
```

```
...
u'currencies': [u'USD'],
u'demonym': u'American Samoan',
u'languages': [u'en', u'sm'],
...
```

Now that we've rolled a couple of our own API consumers, let's take a look at some dedicated libraries that wrap some of the larger web APIs in an easy to use form.

Using Libraries to access Web-APIs

`requests` is capable of negotiating with pretty much all web-APIs and often a little function like Example 5-1 is all you need. But as the APIs start adding authentication and the data structures become more complicated, a good wrapper-library can save a lot of hassle and reduce the tedious book-keeping. In this section I'll cover a couple of the more popular wrapper libraries to give you a feel for the workflow and some useful start-points.

Using Google-spreadsheets

It's becoming more common these days to have live data-sets *in the cloud*. So, for example, you might find yourself required to visualise aspects of a Google-spreadsheet which is the shared data-pool for a group. My preference is to get this data out of the Google-plex and into Pandas, to start exploring it (see ???) but a good library will let you access and adapt the data *in-place*, negotiating the web-traffic as required.

Gspread is the best known Python library for accessing Google-spreadsheets and makes doing so a relative breeze.

You'll need OAth 2.0 credentials to use the API[2]. The most up to date guide can be found here. Following those instructions should provide a JSON file containing your private key.

You'll need to install `gspread` and the latest Python OAuth2 client library. Here's how to do it with `pip`.

```
$ pip install gspread
$ pip install --upgrade oath2client
```

2 OAuth1 access has been deprecated recently.

Depending on your system you may also need PyOpenSSL:

```
$ pip install PyOpenSSL
```

See here for more details and trouble-shooting

> Google's API assumes that the spreadsheets you are trying to access are owned or shared by your API-account, not your personal one. The email-address to share the spreadsheet with is available at your Google developers console and in the JSON credentials key needed to use the API. It should look something like `account-1@My Project...iam.gserviceaccount.com`.

With those libraries installed you should be able to access any of your spreadsheets in a few lines. I'm using the Microbe-scope spreadsheet which you can see here. Example 5-2 shows how to load the spreadsheet.

Example 5-2. Opening a Google-spreadsheet

```
import json
import gspread
from oauth2client.client import SignedJwtAssertionCredentials

json_key = json.load(open('My Project-b8a....json')) ❶
scope = ['https://spreadsheets.google.com/feeds']

credentials = SignedJwtAssertionCredentials(json_key['client_email'],\
    json_key['private_key'].encode(), scope)

gc = gspread.authorize(credentials)

ss = gc.open('Microbe-scope') ❷
```

❶ The JSON credentials file is the one provided by Google-services.

❷ Here we're opening the spreadsheet by name. Alternatives are `open_by_url` or `open_by_id`. See here for details.

Now that we've got our spreadsheet we can see what work-sheets it contains:

```
ss.worksheets()
Out: [<Worksheet 'bugs' id:od6>,
 <Worksheet 'outrageous facts' id:o74cw7y>,
 <Worksheet 'physicians per 1,000' id:okzh6fp>,
 <Worksheet 'amends' id:ogkk64p>]

 ws = ss.worksheet('bugs')
```

With the worksheet *bugs* selected from the spreadsheet, `gspread` allows you to access and change column, row and cell values (assuming the sheet isn't read-only). So we can get the values in the second column with the `col_values` command:

```
ws.col_values(1)
Out: [None,
 'grey = not plotted',
 'Anthrax (untreated)',
 'Bird Flu (H5N1)',
 'Bubonic Plague (untreated)',
 'C.Difficile',
 'Campylobacter',
 'Chicken Pox',
 'Cholera',...
```

Although you can use `gspread's` API to plot directly, using a plot-library like Matplotlib, I prefer to send the whole sheet to Pandas, Python's powerhouse programmatic spreadsheet. This is easily achieved using `gspread's` `get_all_records`, which returns a list of item dictionaries. This list can be used directly to initialise a Pandas `DataFrame` (see ???):

```
df = pd.DataFrame(ws.get_all_records())
df.info()
Out:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 41 entries, 0 to 40
Data columns (total 23 columns):
                                        41 non-null object
average basic reproductive rate         41 non-null object
case fatality rate                      41 non-null object
infectious dose                         41 non-null object
...
upper R0                                41 non-null object
viral load in acute stage               41 non-null object
yearly fatalities                       41 non-null object
dtypes: object(23)
memory usage: 7.7+ KB
```

In ??? we'll see how to interactively explore a DataFrame's data.

Using the Twitter API with Tweepy

The advent of social media has generated a lot of data and an interest in visualising the social-networks, trending hashtags, media-storms etc. contained in it. Twitter's broadcast network is probably the richest source of cool data-visualisations and its API provides tweets[2] filtered by user, hashtag, date etc.

Python's Tweepy is an easy to use Twitter library which provides a number of useful features, such as a `StreamListener` class for streaming live twitter updates. To start using it you'll need a Twitter access token, which can be acquired by following the instructions here to create your twitter application. Once this application is created you can get the keys and access tokens for your app by clicking on the link here.

Tweepy typically requires the four authorisation elements shown here:

```
# The user credential variables to access Twitter API
access_token = "2677230157-Ze3bWuBAw4kwoj4via2dEntU86...TD7z"
access_token_secret = "DxwKAvVzMFLq7WnQGnty49jgJ39Acu...paR8ZH"
consumer_key = "pIorGFGQHShuYQtIxzYWk1jMD"
consumer_secret = "yLc4Hw82G0Zn4vTi4q8pSBcNyHkn35BfIe...oVa4P7R"
```

With those defined, accessing tweets could hardly be easier. Here we create an OAuth `auth` object using our tokens and keys and use it to start an API session. We can then grab the latest tweets from our timeline:

```
import tweepy

auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_token_secret)

api = tweepy.API(auth)

public_tweets = api.home_timeline()
for tweet in public_tweets:
    print tweet.text
```

```
RT @Glinner: Read these tweets https://t.co/QqzJPsDxUD
Volodymyr Bilyachat https://t.co/VIyOHlje6b +1 bmeyer
#javascript
RT @bbcworldservice: If scientists edit genes to
```

2 The free API is currently limited to around 350 requests per hour.

```
make people healthier does it change what it means to be
human?  https://t.co/Vciuyu6BCx h…
RT @ForrestTheWoods:
Launching something pretty cool tomorrow. I'm excited. Keep
...
```

Tweepy's `API` class offers a lot of convenience methods which you can check out here. A common visualisation is using a network graph to show patterns of friends and followers among Twitter subpopulations. The Tweepy methods `follower_ids` (get all users following) and `friends_ids` (get all users being followed) can be used to construct such a network:

```
my_follower_ids = api.follower_ids() ❶

for id in my_followers_ids:
    followers = api.follower_ids(id) ❷
    # ...
```

❶ Gets a list of your followers' ids, e.g. `[1191701545, 1554134420, …]`.

❷ The first argument to `follower_ids` can be an id or screen-name

By mapping followers of followers etc. you can create a network of connections which might just reveal something interesting about groups and subgroups clustered about a particular individual or subject. There's a nice example of just such a twitter analysis here.

One of the coolest features of Tweepy is its `StreamListener` class, which makes it easy to collect and process filtered tweets in real-time. Live updates of twitter streams have been used by many memorable visualisations such as tweetping. Let's set up a little stream to record tweets mentioning Python, JavaScript and Dataviz and save it to a MongoDB database, using the `get_mongo_database` method from "MongoDB" on page 97:

```
# ...
from tweepy.streaming import StreamListener
import json

# ...

class MyStreamListener(StreamListener):
    """ Streams tweets and saves to a MongoDB database """
```

```python
    def __init__(self, api, **kw):
        self.api = api
        super(tweepy.StreamListener, self).__init__()
        self.col = get_mongo_database('tweets', **kw)['tweets'] ❶

    def on_data(self, tweet):
        self.col.insert(json.loads(tweet)) ❷

    def on_error(self, status):
        return True # keep stream open


auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_token_secret)
api = tweepy.API(auth)
stream = tweepy.Stream(auth, MyStreamListener(api))

# Start the stream with track-list of keywords
stream.filter(track=['python', 'javascript', 'dataviz'])
```

❶ The extra kw keywords allow us to pass the MongoDB specific host, port, username/password arguments to the stream-listener.

❷ The data is a raw JSON string that needs decoding before inserting into our *tweets* collection.

Now that we've had a taste of the kind of APIs you might run into in your search for interesting data, let's look at the primary technique you'll use if, as is often the case, no one is providing the data you want in a neat, user-friendly form: Scraping data with Python.

Scraping Data

Scraping is the chief metaphor used for the practice of getting data that wasn't designed to be programmatically consumed off the web. It is a pretty good metaphor because scraping is often about getting the balance right between removing too much and too little. Creating procedures that extract just the right data, as clean as possible, from web-pages is a craft skill and often a fairly messy one at that. But the pay-off is access to visualizable data that often cannot be got in any other way. Approached in the right way scraping can even have an intrinsic satisfaction.

Why we need to scrape

In an ideal virtual world, online data would be organised in a library, with everything catalogued using a sophisticated dewey-decimal system for the web-age. Unfortunately for the keen data hunter, the web has grown organically, often unconstrained by considerations of easy data access for the budding data visualiser. So in reality the web resembles a big mound of data, some of it clean and usable (and thankfully this percentage is increasing) but much of it poorly formed and designed for human consumption. And humans are able to parse the kind of messy, poorly-formed data that our relatively dumb computers have problems with[2].

Scraping is about fashioning selection patterns that grab the data we want and leave the rest behind. If we're lucky the web-pages containing the data have helpful pointers, like named tables, specific identities in preference to generic classes etc.. If we're unlucky then these pointers are missing and we have to resort to using other patterns or, in a worst case, ordinal specifiers such as *third table in the main div*. These latter are obviously pretty fragile, broken in this case if somebody adds a table above the third.

Essentially, if you haven't been given the data in 'clean' form or have access to a web-API to deliver the data you need, in JSON, XML or some other common format, then you will probably find that the dataset you need to create your visualisation is encoded in HTML, in the form of tables, headers, ordered and unordered lists of content and the like. If the data-set is small enough, and we're talking very small, you could resort to cut and paste but, aside from the tedium and inevitable human-error involved, this approach just isn't going to scale. But, generally, although the data is secreted within blocks of HTML those blocks have some repeated structure and, if we're lucky, CSS labels. These two facts allow us to describe, in the formal way a computer understands, where that data is in the HTML. We can then extract it, manually and on a small scale using specialised tools like `requests` and `BeautifulSoup` or in bulk, using `Scrapy` (see ???).

2 Much of modern Machine Learning and Artificial Intelligence research (AI) is dedicated to creating computer software that can cope with messy, noisy, fuzzy, informal data but, as of this book's publication, there's no off-the-shelf solution I know of.

In this section we'll set a little scraping task, aiming to get the some Nobel-prize winners data. We'll use Python's best-of-breed BeautifulSoup for this lightweight scraping foray, saving the heavy guns of Scrapy for the next chapter.

BeautifulSoup and lxml

Python's key lightweight scraping tools are *BeautifulSoup* and *lxml*. Their primary selection syntax is different but, confusingly, both can use each other's parsers. The consensus seems to be that *lxml* is considerably faster but *BeautifulSoup* might be more robust dealing with poorly-formed html. Personally, I've found *lxml* to be robust enough and it's syntax, based on xpaths, more powerful and intuitive. I think for someone coming from web-development, familiar with CSS and Jquery, selection based on CSS is much more natural. But, as mentioned, BeautifulSoup allows us access to these selectors and has a bigger following, which often pays off in, for example, StackOverflow advice. In the following sections I'll use BeautifulSoup's selectors. In the next chapter we'll see *lxml* xpaths selectors in action with Scrapy.

BeautifulSoup is part of the Anaconda packages (see Chapter 1) and easily installed with `pip`:

```
$ pip install beautifulsoup
```

A First Scraping Foray

Armed with requests and BeautifulSoup, let's set ourselves a little task, to get the names, years, categories and nationalities of all the Nobel prize-winners. We'll start at the main Wikipedia Nobel page at *http://en.wikipedia.org/wiki/List_of_Nobel_laureates*. Scrolling down shows a table with all the Laureates by year and category, which is a good start to our minimal data requirements.

Some kind of HTML-explorer is pretty much a must for web-scraping and the best I know is Chrome web-developer's elements tab (see "The Elements Tab" on page 120). Figure 5-1 shows the key-elements involved in quizzing a web-page's structure. We need to know how to select the data of interest, in this case a Wikipedia table, while avoiding other elements on the page. Crafting good selector patterns is the key to effective scraping and highlighting the DOM element using the element inspector gives us both the CSS pattern and, with a right-click on the mouse, the xpath. The latter is

a particularly powerful syntax for DOM-element selection and the basis of our industrial strength scraping solution, *Scrapy*.

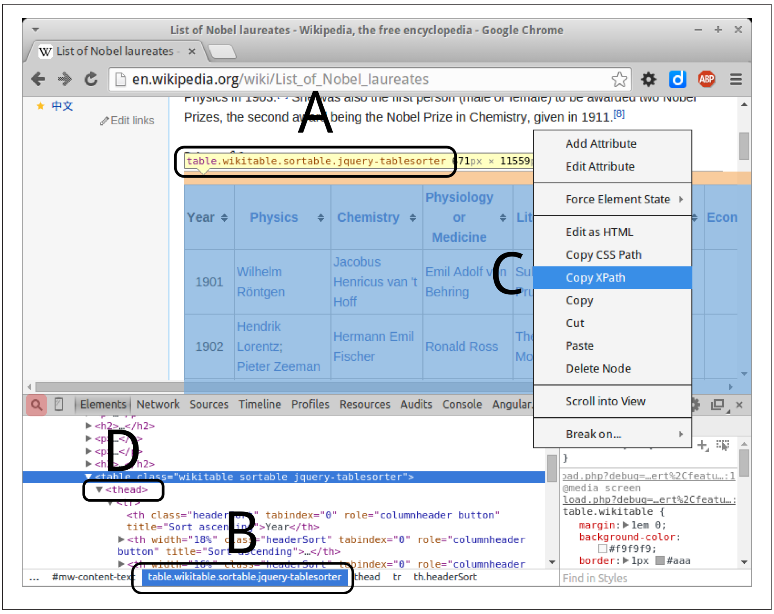


Figure 5-1. Wikipedia's Main Nobel-prize Page: A. and B. show the wikitable's CSS-selector. Right-clicking the mouse and selecting C. (`Copy XPath) gives the table's xpath (*`//[@id="mw-content-text"]/table[1]). D. shows a +`*<thead> tag generated by jQuery.*

Selecting the data

When doing basic scraping on the page's HTML, i.e. not parsing JavaScript-generated pages, we are working with the source code. Sometimes JavaScript is used to alter the page structure, an example being the addition by jQuery of extra tags (see Figure 5-1 D.) to make the table sortable. These are only visible on the rendered page so cannot be used as selection guides working on the raw source. For this reason it's sensible to have the source HTML to hand and be prepared for JS additives. To access the source you can right-click and select from the menu or use the CTRL-U short-cut.

The first thing we need to do is select the data table. The *get_main_table()* method shown in Example 5-3 does just that

Example 5-3. Selecting a Wikipedia table with BeautifulSoup

```
from bs4 import BeautifulSoup
import requests

BASE_URL = 'http://en.wikipedia.org'
# Wikipedia will reject our request unless we add a 'User-Agent'
# attribute to our http header.
HEADERS = {'User-Agent': 'Mozilla/5.0'}

def get_main_table():
    """ Get the wikitable list of Nobel winners """
    # Make a request to the Nobel-page, setting valid headers
    response = requests.get(
        BASE_URL + '/wiki/List_of_Nobel_laureates',
        headers=HEADERS)
    # Parse the response content with BeautifulSoup
    soup = BeautifulSoup(response.content)
    # Use the parsed tree to find our table
    table = soup.find('table', {'class':'wikitable sortable'}) ❶
    return table
```

❶ The second argument to `find` takes a dictionary of element properties: there's only one table with the classes *wikitable* and *sortable*.

First we send the page content to BeautifulSoup, which parses it, creating a tree-structure which we apply selectors to. If you look at Figure 5-1 you'll see that our table has the css classes *wikitable* and *sortable*. There's only one such table on the page so these two classes disambiguate the data table we need. Using the *find* method on our parsed content we pass in a dictionary to filter any child elements by class (in this case), id, name etc..

Let's see what we get, using the selection's *prettify* method to return a readable string.

```
wikitable = get_main_table()
print(wikitable.prettify())
```

```
<table class="wikitable sortable">
 <tr>
  <th>
   Year
  </th>
  <th width="18%">
   <a href="/wiki/List_of_Nobel_laureates_in_Physics" title="List of No...
    Physics
   </a>
```

```
    </th>
    <th width="16%">
     <a href="/wiki/List_of_Nobel_laureates_in_Chemistry" title="List of ...
      Chemistry
      ...

   <tr>
    <td align="center">
     1901
    </td>
    <td>
     <span class="sortkey">
      Röntgen, Wilhelm
     </span>
    ...
  </table>
```

This output shows the parser has successfully constructed an HTML tree from our wiki-table. Now let's get some information from it.

Crafting some selection patterns

Having successfully selected our data table, we now want to craft some selection patterns to scrape the required data. Using the HTML-explorer you can see that the individual winners are contained in `<td>` cells, with an href `<a>` link to Wikipedia's bio-pages (in the case of individuals). Here's a typical target row, with CSS-classes we can use as targets to get the data in the `<td>` cells.

```
   <tr>
    <td align="center">
     1901
    </td>
    <td>
     <span class="sortkey">
      Röntgen, Wilhelm
     </span>
     <span class="vcard">
      <span class="fn">
       <a href="/wiki/Wilhelm_R%C3%B6ntgen" title="Wilhelm Röntgen">
        Wilhelm Röntgen
       </a>
      </span>
     </span>
    </td>
    <td>
    ...
  </tr>
```

If we loop through these data cells, keeping track of their row (year) and column (category) then we should be able to create a list of winners with all the data we specified except nationality.

The following *get_column_titles* function scrapes our table for the Nobel category column headers, ignoring the first *Year* column. Often the header cell to a Wikipedia table contains a web-linked *a* tag and all the Nobel categories fit this model, pointing to their respective Wikipedia pages. If the header is not clickable we store its text and a null href:

```
def get_column_titles(table):
    """ Get the Nobel categories from the table header """
    cols = []
    for th in table.find('tr').find_all('th')[1:]: ❶
        link = th.find('a')
        # Store the category name and any Wikipedia link it has
        if link:
            cols.append({'name':link.text,\
                         'href':link.attrs['href']})
        else:
            cols.append({'name':th.text, 'href':None})
    return cols
```

❶ We loop through the table head, ignoring the first *Year* column ([1:]). This selects the column titles shown in Figure 5-2.

Let's make sure `get_column_titles` is giving us what we want:

```
get_column_titles(wikitable)
Out:
[{'href': '/wiki/List_of_Nobel_laureates_in_Physics',
  'name': u'Physics'},
 {'href': '/wiki/List_of_Nobel_laureates_in_Chemistry',
  'name': u'Chemistry'}, ...
```

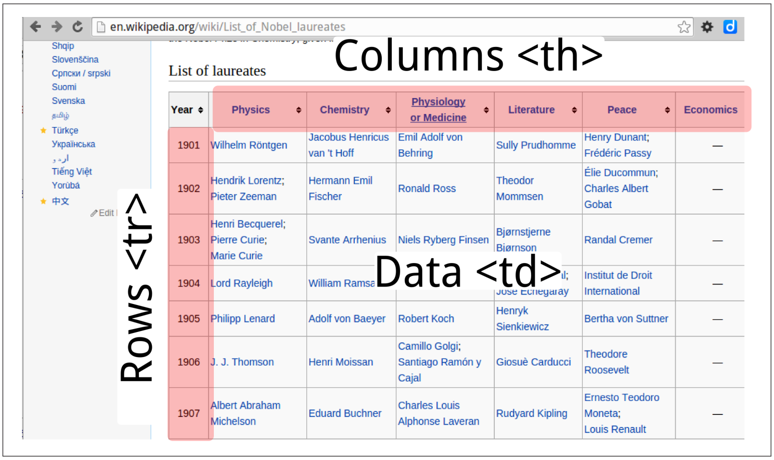


Figure 5-2. Wikipedia's table of Nobel-winners

We use the column names from `get_column_titles` in the following *get_nobel_winners_BS* function:

```
def get_nobel_winners_BS(table):
    cols = get_column_titles(table)
    winners = []
    for row in table.find_all('tr')[1:-1]: ❶
        year = int(row.find('td').text) # Gets 1st <td>
        for i, td in enumerate(row.find_all('td')[1:]): ❷
            for winner in td.find_all('a'):
                href = winner.attrs['href']
                if not href.startswith('#endnote'):
                    winners.append({
                        'year':year,
                        'category':cols[i]['name'],
                        'name':winner.text,
                        'link':winner.attrs['href']
                    })
    return winners
```

❶ Gets all the Year-rows, starting from the second, corresponding to the rows in Figure 5-2.

❷ Finds the `<td>` data-cells shown in Figure 5-2.

Iterating through the year rows, we take the first *Year* column and then iterate over the remaining columns, using *enumerate* to keep track of our index, which will map to the category column names. We know that all the winner names are contained in an <a> tag but

that there are occasional extra <a> tags beginning with *#endnote*, which we filter for. Finally we append a year, category, name and link dict to our data-array. Note that the winner selector has an *attrs* dict containing, among other things, the <a> tag's href.

Let's use our wikitable to confirm that `get_nobel_winners_BS` delivers a list of winner dictionaries with the correct attributes. We'll use Python's built-in `pprint` module to pretty-print the results:

```
from pprint import pprint

wikitable = get_main_table()
winners = get_nobel_winners_BS(wikitable)
pprint(winners[:10])
```

```
[{'category': u'Physics',
  'link': '/wiki/Wilhelm_R%C3%B6ntgen',
  'name': u'Wilhelm R\xf6ntgen',
  'year': 1901},
 {'category': u'Chemistry',
  'link': '/wiki/Jacobus_Henricus_van_%27t_Hoff',
  'name': u"Jacobus Henricus van 't Hoff",
  'year': 1901},
 {'category': u'Physiology\nor Medicine',
  'link': '/wiki/Emil_Adolf_von_Behring',
  'name': u'Emil Adolf von Behring',
  'year': 1901},
...
```

Now that we have the full list of Nobel prize-winners and links to their Wikipedia pages, we will be using these links to scrape data from the individuals' biographies. This will involve making a largish number of requests, and it's not something we really want to do more than once. The sensible thing is to cache the data we scrape, allowing us to try out various scraping experiments without returning to Wikipedia.

Caching the web-pages

It's easy enough to rustle up a quick cacher in Python but as often as not easier still to find a better solution written by someone else and kindly donated to the open-source community. `Requests` has a nice plugin called `requests-cache` which, with a few lines of configuration, will take care of all your basic caching needs.

First we install the plugin using `pip`:

```
$ pip install --upgrade requests-cache
```

requests-cache uses Monkey-patching to dynamically replace parts of the requests API at run-time. This means it can work transparently. You just have to install its cache and then use requests as usual, with all the caching being taken care of. Here's the simplest way to use requests-cache:

```
import requests
import requests_cache

requests_cache.install_cache()
# use requests as usual...
```

The install_cache method has a number of useful options, e.g. allowing you to specify the cache backend (sqlite, memory, mongdb or redis) or set an expiry time (expiry_after) in seconds on the caching. So the following creates a cache named *nobel_pages* with an sqlite backend and pages that expire in two hours (7200s).

```
requests_cache.install_cache('nobel_pages',\
                             backend='sqlite', expire_after=7200)
```

If you get tired trying to calculate the number of seconds for your expiry time, you set the timedelta in install_cache:

```
from datetime import timedelta

expire_after = timedelta(days=1)
requests_cache.install_cache(expire_after=expire_after)
```

requests-cache will serve most of your caching needs and couldn't be much easier to use. For more details see here where you'll also find a little example of request-throttling, a useful technique when doing bulk scraping.

Scraping the winners' nationalities

With caching in place, let's try getting the winners' nationalities, using the first fifty for our experiment. A little *get_nationality()* function will use the winner links we stored earlier to scrape their page and then use the infobox shown in Figure 5-3 to get the *Nationality* attribute.



Figure 5-3. Scraping a winner's nationality

When scraping you are looking for reliable patterns, repeating elements with useful data. As we'll see, the Wikipedia infoboxes for individuals are not such a reliable source but clicking on a few random links certainly gives that impression. Depending on the size of the dataset, it's good to perform a few experimental sanity-checks. You can do this manually but, as mentioned at the start of the chapter, this won't scale or improve your craft skills.

Example 5-4 takes one of the winner dictionaries we scraped earlier and returns a name-labelled dictionary with a *Nationality* key if one were found. Let's run it on the first fifty winners and see how often a *Nationlity* attribute is missing:

Example 5-4. Scraping the winner's country from their biography page

```
def get_nationality(w):
    """ scrape biographic data from the winner's wikipedia page """
    data = get_url('http://en.wikipedia.org/' + w['link'])
    soup = BeautifulSoup(data)
    person_data = {'name': w['name']}
    attr_rows = soup.select('table.infobox tr') ❶
    for tr in attr_rows:                          ❷
        try:
```

```
            attribute = tr.find('th').text
            if attribute == 'Nationality':
                person_data[attribute] = tr.find('td').text
        except AttributeError:
            pass

    return person_data
```

❶ Selects all rows of the infobox table

❷ Cycles through the rows looking for a *Nationality* field.

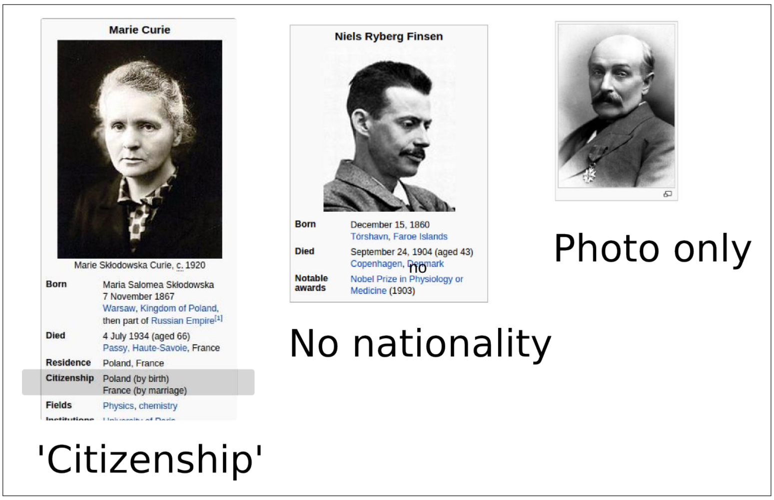


Figure 5-4. Winners without a recorded nationality

Example 5-5 shows 14 of the 50 first winners failed our attempt to scrape their nationality. In the case of the *Institut de Droit International* national affiliation may well be moot but Theodore Roosevelt is about as American as they come. Clicking on a few of the names shows the problem (see Figure 5-4). The lack of a standardised biography format means synonyms for Nationality are often employed, as in Marie Curie's *Citizenship*, sometimes no reference is made, as with Niels Finsen and Randall Cremer has nothing but a photograph in his info-box. We can discard the infoboxes as a reliable source of winners' nationalities but, as they appeared to be the only regular source of potted data, this sends us back to the drawing board. In the next chapter we'll see a successful approach using Scrapy and a different start page.

Example 5-5. Testing for scraped nationalities

```
wdata = []
# test first 50 winners
for w in winners[:50]:
    wdata.append(get_nationality(w))
missing_nationality = []
for w in wdata:
    # if missing 'Nationality' add to list
    if not w.get('Nationality'):
        missing_nationality.append(w)
# output list
missing_nationality

[{'name': u'\xc9lie Ducommun'},
 {'name': u'Charles Albert Gobat'},
 {'name': u'Marie Curie'},
 {'name': u'Niels Ryberg Finsen'},
 {'name': u'Randal Cremer'},
 {'name': u'Institut de Droit International'},
 {'name': u'Bertha von Suttner'},
 {'name': u'Theodore Roosevelt'},
 ...
```

Although Wikipedia is a relative free-for-all, production-wise, where data is designed for human-consumption you can expect a lack of rigour. Many sites have similar gotchas and as the data sets get bigger more tests may be needed to find the flaws in a collection pattern.

Although our first scraping exercise was a little artificial, in order to introduce the tools, I hope it captured something of the slightly messy spirit of web-scraping. The ultimately abortive pursuit of a reliable nationality field for our Nobel data-set could have been forestalled by a bit of web-browsing and manual HTML-source trawling but if the dataset was significantly larger and the failure rate a bit smaller then programmatic detection, which gets easier and easier as you become acquainted with the scraping modules, really starts to deliver.

This little scraping test was designed to introduce BeautifulSoup and shows that collecting the data we set ourselves requires a little more thought, often the case with scraping. In the next chapter we'll wheel out the big gun *Scrapy* and, with what we've learned in this section, harvest the data we need for our Nobel-visualisation.

Summary

In this chapter we've seen examples of the most common ways in which data can be sucked out of the web and into Python containers, databases or Panda's DataSets. Python's `requests` library is the true work-horse of HTTP negotiation and a fundamental tool in our dataviz toolchain. For simpler, RESTful APIs, consuming data with `requests` is a few lines of Python away. For the more awkward APIs, for example those with potentially complicated authorisation, a wrapper library like Tweepy (for twitter) can save a lot of hassle. Decent wrappers can also keep track of access rates and where necessary throttle your requests. This is a key consideration, particularly where there is the possibility of black-listing unfriendly consumers.

We also started our first forays into data-scraping, often a necessary fall-back where no API exists and the data is for human consumption. In the next chapter we'll aim to get all the Nobel-prize data needed for the book's visualisation, using Python's Scrapy, an industrial-strength scraping library.

CHAPTER 6
Heavyweight Scraping with Scrapy

Where BeautifulSoup is a very handy little pen-knife for fast and dirty scraping, Python has a library, Scrapy, which can do large-scale data scrapes with ease. It has all the things you'd expect, like built in caching (with expiration times), asynchronous threading via Python's Twisted web-framework, User-Agent randomisation and a whole lot more. Although it's got a steeper learning curve than *BeautifulSoup*, it's still Python-succinct and quickly becomes routine. For any large data scraping jobs, this is a must-have tool.

In "Scraping Data" on page 160, we managed to scrape a dataset containing all the Nobel-winners by name, year and category. We did a speculative scrape of the winners' linked biography pages which showed that extracting the country of nationality was going to be difficult. In this chapter we'll set the bar on our Nobel-prize data a bit higher and aim to scrape objects of the form shown in Example 6-1.

Example 6-1. Our targeted Nobel JSON object

```
{
 'category': u'Physiology or Medicine',
 'date_of_birth': u'8 October 1927',
 'date_of_death': u'24 March 2002',
 'gender': 'male',
 'link': u'http://en.wikipedia.org/wiki/C%C3%A9sar_Milstein',
 'name': u'C\xe9sar Milstein',
 'country': u'Argentina',
 'place_of_birth': u'Bah\xeda Blanca ,  Argentina',
```

```
 'place_of_death': u'Cambridge , England',
 'year': 1984}
```

In addition to this data, we'll aim to scrape prize winners' photos (where applicable) and some potted biographical data (see Figure 6-1). We'll be using the photos and body-text to add a little character to our Nobel-visualisation.



Figure 6-1. Scraping targets for the prize-winners' pages

Setting up Scrapy

Scrapy is one of the Anaconda packages (see Chapter 1) so you should already have it to hand. If you're not using Anaconda, a quick pip install will do the job[2]:

```
$ pip install scrapy
```

With Scrapy installed, you should have access to the `scrapy` command. Unlike the vast majority of Python libraries, Scrapy is designed to be driven from the command-line, within the context of a scraping project, defined by configuration files, scraping-spiders, pipelines etc. Let's generate a fresh project for our Nobel-prize scraping, using the `startproject` option. This is going to generate a

2 See here for any platform-specific details.

project folder so make sure you run it from a suitable work directory:

```
$ scrapy startproject nobel_winners
New Scrapy project 'nobel_winners' created in:
    /home/kyran/workspace/.../scrapy/nobel_winners

You can start your first spider with:
    cd nobel_winners
    scrapy genspider example example.com
```

As the output of `startproject` says, you'll want to switch to the `nobel_winners` directory in order to start driving scrapy.

Let's take a look at the project's directory tree:

```
nobel_winners
├── nobel_winners
│   ├── __init__.py
│   ├── items.py
│   ├── pipelines.py
│   ├── settings.py
│   └── spiders
│       └── __init__.py
└── scrapy.cfg
```

As shown, the project directory has a sub-directory with the same name and a config file `scrapy.cfg`. The `nobel_winners` sub-directory is a Python module (containing an `__init__.py` file) with a few skeleton files and a `spiders` directory, which will contain your scrapers.

Establishing the Targets

In "Scraping Data" on page 160 we tried to scrape the Nobel winners' nationalities from their biography pages but found they were missing or inconsistently labelled in many cases (see ???). Rather than get the country-data indirectly, a little Wikipedia searching shows a way through. There is a page that lists winners by country. The winners are presented in titled, ordered-lists (see Figure 6-2), not in tabular form, which makes recovering our basic name, category and year data a little harder. Also the data organisation is not ideal, e.g. the header titles and lists aren't in useful, separate blocks. But it still nets us the all-important country field and a few, well-structured xpath queries should easily target the data we need.

Figure 6-2 shows the starting page for our first spider along with the key elements it will be targeting. A list of country name titles (A) are followed by an ordered list (B) of their Nobel-prize winning citizens.

In order to scrape the list-data we need to fire up our Chrome browser's development tools (see "The Elements Tab" on page 120) and inspect the target elements using the *Elements* tab and its inspector (magnifying glass). Figure 6-3 shows the key HTML targets for our first spider: Header titles (h2) containing a country name and followed by an ordered list (ol) of winners (lis).

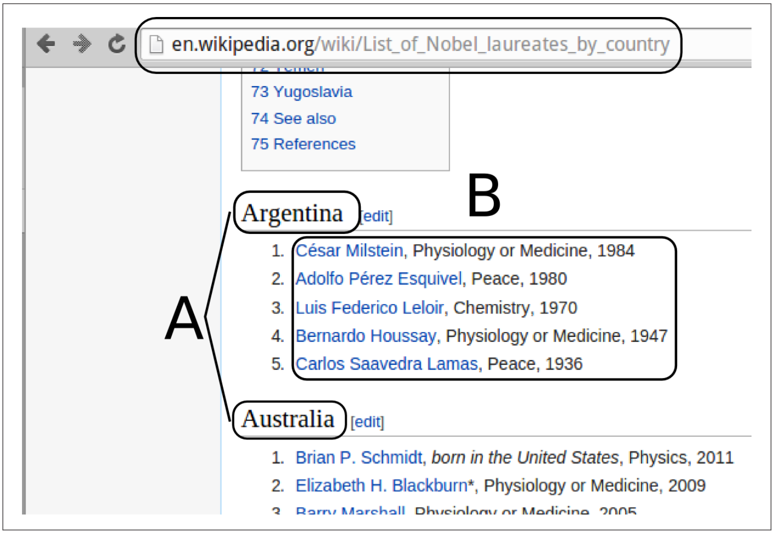


Figure 6-2. Scraping Wikipedia's Nobel-prizes by nationionality



Figure 6-3. Finding the HTML-targets for the Wiki-list

Targeting HTML with Xpaths

Scrapy uses the xpaths to define its HTML targets. Xpath is a syntax for describing parts of an X(HT)ML document and while it can get rather complicated, the basics are straightforward and will often solve the job at hand.

You can get the xpath of an HTML element by using Chrome's Elements tab to hover over the source and then right-clicking the mouse and selecting *Copy Xpath*. For example, in the case of our Nobel Wiki-list's country names (h2 in Figure 6-3) , selecting the xpath of 'Argentina' (the first country) gives the following:

```
//*[@id="mw-content-text"]/h2[1]
```

We can use the following xpath rules to decode it:

//E	Element <E> by relative reference (in this case relative to the root Document)
//E[@id="foo"]	select Element <E> with id *foo*
//*[@id="foo"]	select any element with id *foo*
//E/F[1]	first child element <F> of element <E>
//E/*[1]	first child of element <E>

Following these rules shows our Argentinian title `//*[@id="mw-content-text"]/h2[1]` is the first header (h2) child of a DOM element with id `mw-content-text`. This is equivalent to the following HTML:

```
<div id="mw-content-text">
    <h2>
        ...
    </h2>
    ...
</div>
```

Note that unlike Python, the xpaths don't use a zero-based index but make the first member '1'.

Testing xpaths with the Scrapy shell

Getting your xpath targeting right is crucial to good scraping and can involve a degree of iteration. Scrapy makes this process much easier by providing a command-line shell, which takes a URL and creates a response context in which you can try out your xpaths, like so:

```
$ scrapy shell https://en.wikip...List_of_Nobel_laureates_by_country

2015-12-15 17:42:12+0000 [scrapy] INFO: Scrapy 0.24.4 started
(bot: nobel_winners)
...
2015-12-15 17:42:12+0000 [default] INFO: Spider opened
2015-12-15 17:42:13+0000 [default] DEBUG: Crawled (200)
<GET https://en.wikip...List_of_Nobel_laureates_by_country>
(referer: None)
[s] Available Scrapy objects:

[s]   crawler  <scrapy.crawler.Crawler object at 0x3a8f510>
[s]   item {}
[s]   request    <GET https://en.wik...Nobel_laureates_by_country>
[s]   response   <200 https://en.wik...Nobel_laureates_by_country>
[s]   settings   <scrapy.settings.Settings object at 0x34a98d0>
[s]   spider     <Spider 'default' at 0x3f59190>

[s] Useful shortcuts:
[s]   shelp()    Shell help (print this help)
[s]   fetch(req_or_url) Fetch request (or URL) and update local
objects
[s]   view(response)    View response in a browser

In [1]:
```

Now we have an IPython-based shell with code-complete and syntax highlighting, in which to try out our xpath targeting. Let's grab all the `<h2>` headers on the Wiki-page:

```
In [1]: h2s = response.xpath('//h2')
```

The resulting `h2s` is a SelectorList, a specialised Python `list` object. Let's see how many headers we have:

```
len(h2s)
Out:
76
```

We can grab the first Selector object and query its methods using tab auto-complete:

```
h2 = h2s[0]
h2.
h2.css              h2.namespaces       h2.remove_namespaces
h2.text             h2.extract          h2.re
h2.response         h2.type             h2.register_namespace  h2.select
```

You'll often be using the `extract` method to get the raw result of the xpath selector:

```
h2.extract()
Out:
u'<h2>Contents</h2>'
```

This shows our first `<h2>` header is that of the table of contents for our list of winners by nationality. Let's look at the second header:

```
h2s[1].extract()

u'<h2>
  <span class="mw-headline" id="Argentina">Argentina</span>
  <span class="mw-editsection"><span class="mw-editsection-bracket">
  ...
  </h2>'
```

This shows that our country headers start on the second `<h2>` and contain a `span` with class `mw-headline`. We can use the presence of the `mw-headline` class as a filter for our country headers and the contents as our country label. Let's try out an xpath, using the selector's `text` method to extract the text from the `mw-headline` span. Note that we use the `xpath` method of the `<h2>` selector, which makes the xpath query relative to that element.

```
h2_arg = h2s[1]
country = h2_arg.xpath('span[@class="mw-headline"]/text()').extract()
country
```

```
Out:
[u'Argentina']
```

The `extract` method returns a list of possible matches, in our case the single *Argentina* string. By iterating through the `h2s` list, we can now get our country names.

Assuming we have a country's `<h2>` header, we now need to get the `<ol>` ordered list of Nobel winners following it (Figure 6-2 B.). Handily the xpath `following-sibling` selector can do just that. Let's grab the first ordered list after the Argentina header:

```
ol_arg = h2_arg.xpath('following-sibling::ol[1]')
ol_arg
Out:
[<Selector xpath='following-sibling::ol[1]' data=u'<ol>\n<li>
<a href="/wiki/C%C3%A9sar_Milst'>]
```

Looking at the truncated data for `ol_arg` shows we have selected an ordered-list. Note that even though there's only one `Selector`, `xpath` still returns a `SelectorList`. For convenience you'll generally just select the first member directly:

```
ol_arg = h2_arg.xpath('following-sibling::ol[1]')[0]
```

Now that we've got the ordered list, lets get a list of its member `<li>` elements:

```
lis_arg = ol_arg.xpath('li')
len(lis_arg)
Out: 5
```

Let's examine one of those list elements using `extract`. As a first test, we're looking to scrape the name of the winner and capture the list element's text.

```
li = lis_arg[0] # select the first list element
li.extract()
Out:
u'<li><a href="/wiki/C%C3%A9sar_Milstein"
         title="C\xe9sar Milstein">C\xe9sar Milstein</a>,
         Physiology or Medicine, 1984</li>'
```

Extracting the list element shows a standard pattern: A hyperlinked name to the winner's Wikipedia page followed by a comma-separated winning category and year. A robust way to get the winning name is just to select the text of the list element's first `<a>` tag:

```
name = li.xpath('a//text()')[0].extract()
name
Out: u'C\xe9sar Milstein'
```

It's often useful to get all the text in, for example, a list element, stripping the various HTML <a>, <span> etc. tags. descendent-or-self gives us a handy way of doing this, producing a list of the descendents' text:

```
list_text = li.xpath('descendant-or-self::text()').extract()
list_text
Out: [u'C\xe9sar Milstein', u', Physiology or Medicine, 1984']
```

We can get the full text by joining the list elements together:

```
' '.join(list_text)
Out: u'C\xe9sar Milstein , Physiology or Medicine, 1984'
```

Note that the first item of list_text is the winner's name, giving us another way to access it if, for example, it was missing a hyperlink.

Now that we've established the xpaths to our scraping targets (the name and link-text of the Nobel winners) let's incorporate them into our first Scrapy spider.

> Getting the right Xpath expression for your target element(s) can be a little tricky and those difficult edge cases can demand a complex nest of clauses. The use of a well-written cheat-sheet can be a great help here and thankfully there are many good xpath ones. A very nice selection can be found here with this color-coded one being particularly useful.

A First Scrapy Spider

Armed with a little Xpath knowledge, let's produce our first scraper, aiming to get the country and link-text for the winners (Figure 6-2 A. and B.).

Scrapy calls its scrapers *spiders*, each of which is a Python module placed in the spiders directory of your project. We'll call our first scraper nwinner_list_spider.py:

```
.
├── nobel_winners
│   ├── __init__.py
│   ├── items.py
```

```
|   ├── pipelines.py
|   ├── settings.py
|   └── spiders
|       |─ __init__.py
|       └── nwinners_list_spider.py <---
└── scrapy.cfg
```

Spiders are sub-classed `scrapy.Spider` classes and any placed in the `spiders` directory will be automatically detected by Scrapy and made accessible by name to the `scrapy` command.

The basic Scrapy spider shown in Example 6-2 follows a pattern you'll be using with most of your spiders. First you subclass a Scrapy `item` to create fields for your scraped data (section A inExample 6-2). You then create a named spider by subclassing `scrapy.Spider` (section B in Example 6-2). The spider's name will be used when calling `scrapy` from the command line. Each spider has a `parse` method which deals with the HTTP requests to a list of start URLs contained in a `start_url` class attribute. In our case the start URL is the Wikipedia page for Nobel laureates by country.

Example 6-2. A first Scrapy spider

```
# nwinners_list_spider.py

import scrapy
import re
# A. Define the data to be scraped
class NWinnerItem(scrapy.Item):
    country = scrapy.Field()
    name = scrapy.Field()
    link_text = scrapy.Field()

# B Create a named spider
class NWinnerSpider(scrapy.Spider):
    """ Scrapes the country and link-text of the Nobel-winners. """

    name = 'nwinners_list'
    allowed_domains = ['en.wikipedia.org']
    start_urls = [
        "http://en.wikipedia.org ... of_Nobel_laureates_by_country"
    ]
    # C A parse method to deal with the HTTP response
     def parse(self, response):

        h2s = response.xpath('//h2') ❶

        items = []
```

```
    for h2 in h2s:
        country = h2.xpath('span[@class="mw-headline"]/text()')\ ❷
        .extract()
        if country:
            winners = h2.xpath('following-sibling::ol[1]') ❸
            for w in winners.xpath('li'):
                text = w.xpath('descendant-or-self::text()')\
                .extract()
                items.append(NWinnerItem(
                    country=country[0], name=text[0],
                    link_text = ' '.join(text)
                    ))
    return items
```

❶ Gets all the `<h2>` headers on the page, most of which will be our target country titles.

❷ Where possible, gets the text of the `<h2>` element's child `<span>` with class `mw-headline`.

❸ Gets the list of country winners.

The `parse` method in Example 6-2 receives the response from an HTTP requests to the Wikipedia Nobel page and is required to return a list containing Scrapy items. These items are then converted to a list of JSON objects which can be saved to an output file.

Let's run our first spider to make sure we're correctly parsing and scraping our Nobel data. First navigate to the `nobel_winners` root directory (containing the `scrapy.cfg` file) of the scraping project. Let's see what scraping spiders are available:

```
$ scrapy list
nwinners_list
```

As expected, we have one `nwinners_list` spider sitting in the `spiders` directory. To start it scraping we use the `crawl` command and direct the output to a `nwinners.json` file. By default we will get a lot of Python logging information accompanying the crawl:

```
$ scrapy crawl nwinners_list -o nobel_winners.json
2015- ... [scrapy] INFO: Scrapy 0.24.4 started (bot: nobel_winners)
2015- ... [scrapy] INFO: Optional features available: ssl, http11
...
2015- ... [nwinners_list] INFO: Closing spider (finished)
2015- ... [nwinners_list] INFO: Dumping Scrapy stats:
        {'downloader/request_bytes': 551,
```

```
             'downloader/request_count': 2,
             'downloader/request_method_count/GET': 2,
             'downloader/response_bytes': 45469,
             ...
             'item_scraped_count': 1075, ❶
2015- ...  [nwinners_list] INFO: Spider closed (finished)
```

❶ We scraped 1075 Nobel winners from the page.

The output of the scrapy `crawl` shows 1075 items successfully scraped. Let's look at our JSON output file to make sure things have gone according to plan:

```
$ head nobel_winners.json
[{"country": "Argentina",
  "link_text": "C\u00e9sar Milstein , Physiology or Medicine, 1984",
  "name": "C\u00e9sar Milstein"},
 {"country": "Argentina",
  "link_text": "Adolfo P\u00e9rez Esquivel , Peace, 1980",
  "name": "Adolfo P\u00e9rez Esquivel"},
  ...
```

As you can see, we have an array of JSON objects with the four key fields present and correct.

Now we have a spider that successfully scrapes the list-data for all the Nobel winners on the page, let's start refining it to grab all the data we are targeting for our Nobel visualisation (see Example 6-1 and Figure 6-1).

First, let's add all the data we plan to scrape as fields to our `scrapy.Item`:

```
...
class NWinnerItem(scrapy.Item):
    name = scrapy.Field()
    link = scrapy.Field()
    year = scrapy.Field()
    category = scrapy.Field()
    nationality = scrapy.Field()
    gender = scrapy.Field()
    born_in = scrapy.Field()
    date_of_birth = scrapy.Field()
    date_of_death = scrapy.Field()
    place_of_birth = scrapy.Field()
    place_of_death = scrapy.Field()
    text = scrapy.Field()
...
```

It's also sensible to simplify the code a bit and use a dedicated function, `process_winner_li` to process the winners' link-text. We'll pass a link selector and country name to it and return a dictionary containing the scraped data:

```
...
    def parse(self, response):

        h2s = response.xpath('//h2')

        items = []
        for h2 in h2s:
            country = h2.xpath('span[@class="mw-headline"]/text()').extract()
            if country:
                winners = h2.xpath('following-sibling::ol[1]')
                for w in winners.xpath('li'):
                    wdata = process_winner_li(w, country[0])
                    ...
```

The `process_winner_li` method is shown in Example 6-3

Example 6-3. Processing a winner's list item

```
# ...
import re
BASE_URL = 'http://en.wikipedia.org'
# ...
def process_winner_li(w, country=None):
    """
    Process a winner's <li> tag, adding country of birth or
    nationality, as applicable.
    """
    wdata = {}

    wdata['link'] = BASE_URL + w.xpath('a/@href').extract()[0] ❶

    text = ' '.join(w.xpath('descendant-or-self::text()')
        .extract())
    # get comma-delineated name and strip trailing white-space
    wdata['name'] = text.split(',')[0].strip()

    year = re.findall('\d{4}', text) ❷
    if year:
        wdata['year'] = int(year[0])
    else:
        wdata['year'] = 0
        print('Oops, no year in ', text)

    category = re.findall(
            'Physics|Chemistry|Physiology or Medicine|Literature|Peace|Economics',
```

```
                  text) ❸
    if category:
        wdata['category'] = category[0]
    else:
        wdata['category'] = ''
        print('Oops, no category in ', text)

    if country:
        if text.find('*') != -1: ❹
            wdata['nationality'] = ''
            wdata['born_in'] = country
        else:
            wdata['nationality'] = country
            wdata['born_in'] = ''

    # store a copy of the link's text-string for any manual corrections
    wdata['text'] = text
    return wdata
```

❶ To grab the *href* attribute from the list-item's `<a>` tag (<li>[winner name]...) , we use the xpath attribute referent @.

❷ Here we use `re`, Python's built-in regex library, to find the four-digit year strings in the list-item's text.

❸ Another use of the regex library to find the Nobel prize category in the text.

❹ An asterisk following the winner's name is used to indicate that the country is the winner's by birth not nationality at the time of the prize. e.g. "William Lawrence Bragg*, Physics, 1915" in the list for Austrialia.

Embracing regex

> Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.
>
> —Jamie Zawinskie

The above quote is a hoary old classic but does sum up what many people feel about regular expressions (regex). There is something of the *Alien Hieroglyphics* about them and tales abound of recursive patterns gone horribly wrong. But the fact is that web-scraping is often about pattern matching messy and under-specified data and

regex is pretty much tailor-made for many of the jobs that crop up. You can probably hack your way around them but embracing them a little will make your life that much easier and the good news is that a little goes a long way. See Example 6-3 for some examples.

Example 6-3 returns all the winners' data available on the main Wikipedia Nobels by Nationality page, i.e. the name, year, category, country (of birth or when awarded the prize) and a link to the individual winners' pages. We'll need to use this last information to get those biographical pages and use them to scrape our remaining target data (see Example 6-1 and Figure 6-1).

Scraping the Individual Biography Pages

The main Wikipedia Nobels by Nationality page gave us a lot of our target data, but winner's date of birth, date of death (where applicable) and gender are still to be scraped. This information is hopefully available, either implicitly or explicitly, on their biography pages (for non-organization winners). Now's a good time to fire up Chrome's Elements tab and take a look at those pages, to work out how we're going to extract the desired data.

We saw in the last chapter (???) that the visible information boxes on individual's pages are not a reliable source of information and are often missing entirely. Until recently[2], a hidden `persondata` table (see Figure 6-4) gave fairly reliable access to such information as place-of-birth, date-of-death etc. Unfortunately this handy resource has been deprecated[3]. The good news is that this is part of an attempt to improve the categorisation of biographical information by giving it a dedicated space in Wikidata, Wikipedia's central storage for its structured data.

2 The author got stung by this removal…

3 See here for an insight into Wikipedia dispute management.



```
</table>
<table id="persondata" class="persondata noprint" style="border:1px solid #a
<tr>
<th colspan="2"><a href="/wiki/Wikipedia:Persondata" title="Wikipedia:Person
</tr>
<tr>
<td class="persondata-label" style="color:#aaa;">Name</td>
<td>Röntgen, Wilhelm</td>
</tr>
<tr>
<td class="persondata-label" style="color:#aaa;">Alternative names</td>
<td>Conrad</td>
</tr>
```

Figure 6-4. A Nobel winner's hidden persondata table

Examining Wikipedia's biography pages with Chrome's Elements tab shows a link to the relevant Wikidata item (see Figure 6-5), which takes you to the biographical data held at `wikidata.org`. By following this link we can scrape whatever we find there, hopefully the bulk of our target data significant dates and places (see Example 6-1).

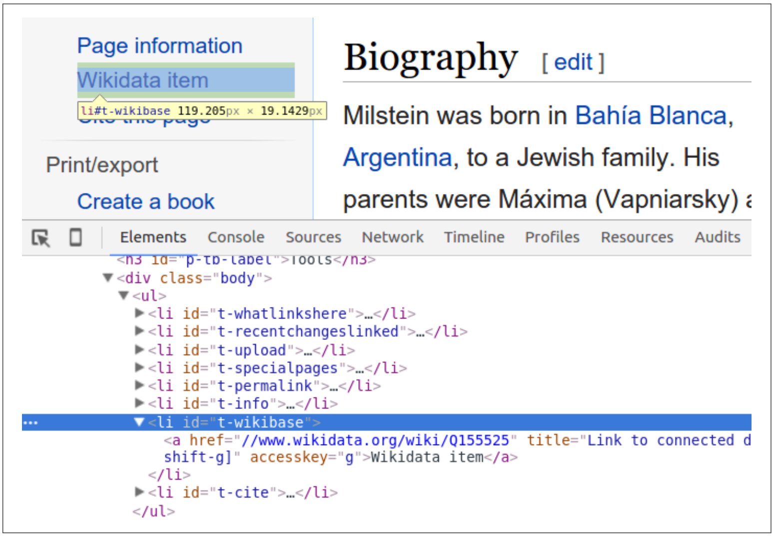


Figure 6-5. Hyperlink to the winner's wikidata

Following the link to Wikidata shows a page containing fields for the data we are looking for, for example the date-of-birth of our prize winner. As Figure 6-6 shows, the properties are embedded in a nest of computer generated HTML, with related codes, which we can use as a scraping identifier. e.g. data-of-birth has the code `P569`.



Figure 6-6. Biographical properties at Wikidata

As Figure 6-7 shows, the actual data we want, in this case a date-string, is contained in a further nested branch of HTML, within its respective property tag. By selecting the `div` and right-clicking we can store the element's xpath and use that to tell Scrapy how to get the data it contains.

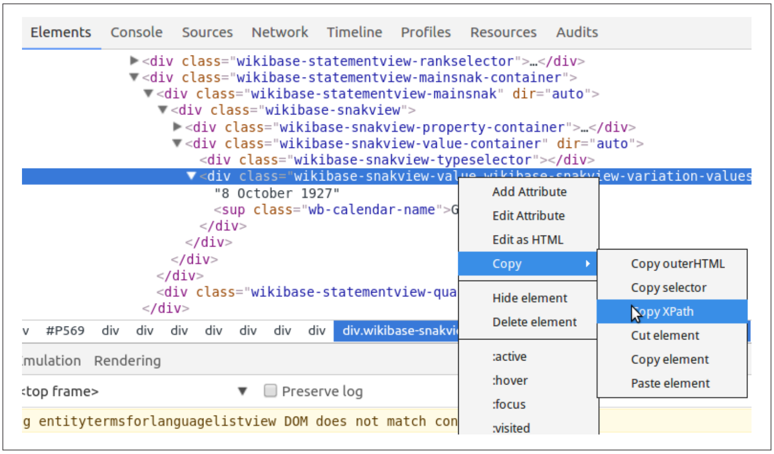


Figure 6-7. Getting the xpath for a Wikidata property

Now we have the xpaths necessary to find our scraping targets, let's put it all together and see how Scrapy chains requests, allowing for complex, multi-page scraping operations.

Chaining Requests and Yielding Data

In this section we'll see how to chain Scrapy requests, allowing us to follow hyper-links, scraping data as we go. First let's enable Scrapy's page-caching. While experimenting with xpath targets etc. we want to limit the number of calls to Wikipedia and it's good manners to be storing our fetched pages. Unlike some data-sets out there, our Nobel-prizewinners' changes but once a year[2].

Caching our pages

As you might expect, Scrapy has a sophisticated caching system that gives you fine-grained control over your page-caching, e.g. allowing you to choose between database or filesystem storage backends, how long before your pages are expired etc.. It is implemented as middleware enabled in our project's `settings.py` module. There are various options available but for the purposes of our Nobel scraping, simply setting `HTTPCACHE_ENABLED` to `True` will suffice:

```
# -*- coding: utf-8 -*-

# Scrapy settings for nobel_winners project
#
# For simplicity, this file contains only the most important settings by
# default. All the other settings are documented here:
#
#     http://doc.scrapy.org/en/latest/topics/settings.html
#

BOT_NAME = 'nobel_winners'

SPIDER_MODULES = ['nobel_winners.spiders']
NEWSPIDER_MODULE = 'nobel_winners.spiders'

# Crawl responsibly by identifying yourself (and your website) on the user-agent
#USER_AGENT = 'nobel_winners (+http://www.yourdomain.com)'

HTTPCACHE_ENABLED = True
```

Check out the full range of Scrapy middleware here.

Having ticked the caching box, let's see how to chain Scray requests.

2 Strictly speaking there are edits being made continually by the Wikipedia community but the fundamental details should be stable until the next set of prizes.

Yielding requests

Our existing spider's `parse` method cycles through the Nobel winners, using the `process_winner_li` method to scrape the country, name, year and category and biography-hyperlink fields. We now want to use the biography hyperlinks to generate a Scrapy request which will fetch the bio-pages and send them to a custom-method for scraping.

Scrapy implements a Pythonic pattern for chaining requests, using Python's `yield` statement to create a generator[2], allowing Scrapy to easily consume any extra page-requests we make. Example 6-4 shows the pattern in action.

Example 6-4. Yielding a request with Scrapy

```
class NWinnerSpider(scrapy.Spider):
    name = 'nwinners_full'
    allowed_domains = ['en.wikipedia.org']
    start_urls = [
        "http://en.wikipedia.org/wiki..._Nobel_laureates_by_country"
    ]

    def parse(self, response):
        filename = response.url.split('/')[-1]
        h2s = response.xpath('//h2')
        items = []
        for h2 in list(h2s)[:2]:
            country = h2.xpath('span[@class="mw-headline"]/text()')
                      .extract()
            if country:
                winners = h2.xpath('following-sibling::ol[1]')
                for w in winners.xpath('li'):
                    wdata = process_winner_li(w, country[0])
                    request = scrapy.Request( ❶
                        wdata['link'],
                        callback=self.parse_bio, ❷
                        dont_filter=True)
                    request.meta['item'] = NWinnerItem(**wdata) ❸
                    yield request ❹

    def parse_bio(self, response):
        item = response.meta['item'] ❺
        ...
```

2 See here for a nice run-down of Python generators and the use of `yield`.

1. Makes a request to the winner's biography page, using the link (wdata[*link*]) scraped from process_winner_li.

2. Sets the callback function to handle the response.

3. Creates a Scrapy Item to hold our Nobel-data and initialises it with the data just scraped from process_winner_li. This Item-data is attached to the meta-data of the request to allow any response access to it.

4. By yielding the request, we make the parse method a generator of consumable requests.

5. This method handles the callback from our bio-link request. In order to add scraped data to our Scrapy Item, we first retrieve it from the response meta-data.

Our investigation of the Wikipedia pages in "Scraping the Individual Biography Pages" on page 189 showed that we need to locate a winner's Wikidata link from their biography-page and use it to generate a request. We will then scrape the date, place, and gender data from the response.

Example 6-5 shows parse_bio and parse_wikidata, the two methods used to scrape our winners' biographical data. parse_bio uses the scraped Wikidata link to request the Wikidata page, yielding the request as it in turn was yielded in the parse method. At the end of the request chain, parse_wikidata retrieves the item and fills in any of the fields available from Wikidata, eventually yielding the item to Scrapy.

Example 6-5. Parsing the winners' biography data

```
# ...

    def parse_bio(self, response):

        item = response.meta['item']
        href = response.xpath("//li[@id='t-wikibase']/a/@href") ❶
                .extract()
        if href:
            request = scrapy.Request('https:' + href[0],\
                           callback=self.parse_wikidata,\ ❷
```

```
                        dont_filter=True)
        request.meta['item'] = item
        yield request


    def parse_wikidata(self, response):
        item = response.meta['item']
        property_codes = [ ❸
            {'name':'date_of_birth', 'code':'P569'},
            {'name':'date_of_death', 'code':'P570'},
            {'name':'place_of_birth', 'code':'P19', 'link':True},
            {'name':'place_of_death', 'code':'P20', 'link':True},
            {'name':'gender', 'code':'P21', 'link':True}
        ]

        p_template = '//*[@id="%(code)s"]/div[2]/div/div/div[2]
                      /div[1]/div/div[2]/div[2]' ❹

        for prop in property_codes:

            extra_html = ''
            if prop.get('link'): # property string in <a> tag
                extra_html = '/a'
            sel = response.xpath(p_template%prop + extra_html + '/text()')
            if sel:
                item[prop['name']] = sel[0].extract()

        yield item ❺
```

❶ Extracts the link to Wikidata identified in Figure 6-5.

❷ Uses the Wikidata link to generate a request with our spider's `parse_wikidata` as a callback to deal with the response.

❸ These are the property codes we found earlier (see Figure 6-6), with names corresponding to fields in our Scrapy Item, `NWinner Item`. Those with a `True` *link* attribute are contained in `<a>` tags.

❹ The nasty, nested xpath for the Wikidata properties used to create this template comes straight from the Chrome's Elements tab (see Figure 6-7).

❺ Finally we yield the item which at this point should have all the target data available from Wikipedia.

With our request chain in place, let's check that the spider is scraping our required data:

```
$ scrapy crawl nwinners_full
2015-... [scrapy] ... started (bot: nobel_winners)
...
2015-... [nwinners_full] DEBUG: Scraped from
         <200 https://www.wikidata.org/wiki/Q155525>
  {'born_in': '',
   'category': u'Physiology or Medicine',
   'date_of_birth': u'8 October 1927',
   'date_of_death': u'24 March 2002',
   'gender': u'male',
   'link': u'http://en.wikipedia.org/wiki/C%C3%A9sar_Milstein',
   'name': u'C\xe9sar Milstein',
   'nationality': u'Argentina',
   'place_of_birth': u'Bah\xeda Blanca',
   'place_of_death': u'Cambridge',
   'text': u'C\xe9sar Milstein , Physiology or Medicine, 1984',
   'year': 1984}
2015-... [nwinners_full] DEBUG: Scraped from
         <200 https://www.wikidata.org/wiki/Q193672>
 {'born_in': '',
  'category': u'Peace',
  'date_of_birth': u'1 November 1878',
  'date_of_death': u'5 May 1959',
  'gender': u'male',
  'link': u'http://en.wikipedia.org/wiki/Carlos_Saavedra_Lamas',
  ...
```

Things are looking good. With the exception of the *born_in* field, which is dependent on a name in the main Wikipedia Nobel winners list having an asterisk, we're getting all the data we were targeting. This data-set is now ready to be cleaned by Pandas in the coming chapter.

Now that we've scraped our basic biographical data for the Nobel-winners, let's go scrape our remaining targets, some biographical body-text, and a picture of the great man or woman, where available.

Scrapy Pipelines

In order to add a little personality to our Nobel visualisation it would be good to have a little biographical text and an image of the winner. Wikipedia's biographical pages generally provide these things so let's go about scraping them.

Up to now our scraped data has been text strings. In order to scrape images, in their various formats, we need to use a Scrapy *pipeline*. Pipelines provide a way of post-processing the items we have scra-

ped and you can define any number of them. You can write your own or take advantage of those already provided by Scrapy, such as the `ImagesPipeline` we'll be using.

In its simplest form, a `pipeline` need only define a `process_item` method. This receives the scraped items and the spider object. Let's write a little pipeline to reject genderless Nobel winners (so we can omit prizes given to organizations rather than individuals) using our existing *nwinners_full* spider to deliver the items. First we add a `DropNonPersons` pipeline to the `pipelines.py` module of our project:

```
# nobel_winners/nobel_winners/settings.py

# Define your item pipelines here
#
# Don't forget to add your pipeline to the ITEM_PIPELINES setting
# See: http://doc.scrapy.org/en/latest/topics/item-pipeline.html

from scrapy.exceptions import DropItem


class DropNonPersons(object):
    """ Remove non-person winners """

    def process_item(self, item, spider):
        if not item['gender']:                    ❶
            raise DropItem("No gender for %s"%item['name'])
        return item ❷
```

❶ If our scraped item failed to find a gender property at Wikidata it is probably an organization such as the Red Cross.. Our visualisation is focused on invdividual winners so here we use `DropItem` to remove the item from our output stream.

❷ We need to return the item to further pipelines or for saving by Scrapy.

As mentioned in the `pipelines.py` header, in order to add this pipeline to the spiders of our project we need to register it in the `settings.py` module by adding it to a `dict` of pipelines and setting it to active (1):

```
# nobel_winners/nobel_winners/settings.py

BOT_NAME = 'nobel_winners'
SPIDER_MODULES = ['nobel_winners.spiders']
```

```
NEWSPIDER_MODULE = 'nobel_winners.spiders'

HTTPCACHE_ENABLED = True
ITEM_PIPELINES = {'nobel_winners.pipelines.DropNonPersons':1}
```

Now we've got the basic workflow for our pipelines, let's add a useful one to our project.

Scraping Text and Images with a Pipeline

We now want to scrape the winners' biography and photos (see Figure 6-1), where available. The biographical text can be scraped using the same method as our last spider but the photos are best dealt with by an image pipeline.

We could easily write our own pipeline to take a scraped image URL, request it from Wikipedia, and save to disk but to do it properly would requires a bit of care. For example, we would like to avoid reloading an image that was recently downloaded or hasn't changed in the meantime. Some flexibility in specifying where to store the images is a useful feature. It would also be good to have the option of converting the images into a common format (e.g. JPG or PNG) or of generating thumbnails. Luckily, Scrapy provides an `ImagesPipeline` object with all this functionality and more. This is one of its media pipelines, which includes a `FilesPipeline` for dealing with general files.

We could add the image and biography-text scraping to our existing `nwinners_full` spider but that's starting to get a little large and segregating this character data from the more formal categories makes sense. So we'll create a new spider, called *nwinners_minibio* which will reuse parts of the previous spider's `parse` method in order to loop through the Nobel winners.

As usual when creating a Scrapy spider, our first job is to get the xpaths for our scraping targets — in this case, where available that's the first part of the winners' biographical text and a photograph of them. To do this we fire up Chrome Elements and explore the HTML source of the biography pages looking for the targets shown in Figure 6-8.

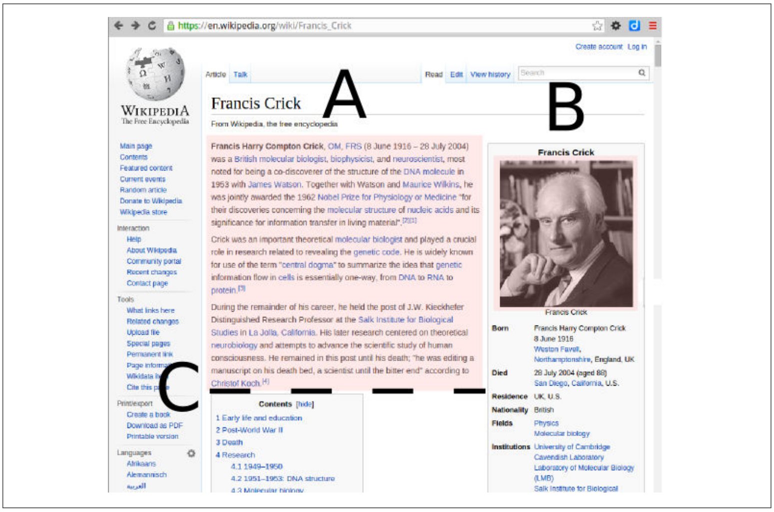


Figure 6-8. The target elements for our biography scraping: The first part of the biography (A), marked by a stop-point (B) and the winner's photograph ©

Investigating with Chrome Elements shows the biographical text (Figure 6-8 A.) is contained in the first paragraphs of the `<div>` with id `mw-content-text`, captured by the xpath `//*[@id="mw-content-text"]/p`. There is an empty paragraph which signals the stop-point (Figure 6-8 B.) of the first section of the biography:

```
<div id="mw-content-text">
  ...
  <p>...</p>
  <p>...</p>
  <p><p> <---- stop-point
  ...
</div>
```

The exploration shows that the photos (Figure 6-8 C.) are contained in a table of class `infobox`, being the only image in that table:

```
<table class="infobox vcard">
  ...
        <img alt="Francis Crick crop.jpg" src="//upload... />
  ...
</table>
```

The xpath `//table[contains(@class,"infobox")]//img/@src` will get the source address of the image.

As with our first spider, we first need to declare a Scrapy `Item` to hold our scraped data. We'll scrape the bio-link and name of the winner, which we can use as identifiers for the image and text. We also need somewhere to store our image-urls (although we will only scrape one bio-image, I'll cover the multiple-image use-case), the resultant images references (a file-path), and a `bio_image` field to store the particular image we're interested in:

```
import scrapy
import re

BASE_URL = 'http://en.wikipedia.org'


class NWinnerItem(scrapy.Item):
    link = scrapy.Field()
    name = scrapy.Field()
    mini_bio = scrapy.Field()
    image_urls = scrapy.Field()
    bio_image = scrapy.Field()
    images = scrapy.Field()
...
```

Now we reuse the scraping loop over our Nobel winners (see Example 6-4 for details), this time yielding a request to our new `get_mini_bio` method, which will scrape the image-urls and bio-text:

```
class NWinnerSpider(scrapy.Spider):
    name = 'nwinners_minibio'
    allowed_domains = ['en.wikipedia.org']
    start_urls = [
        "http://en.wikipedia.org/...of_Nobel_laureates_by_country"
    ]

    def parse(self, response):

        filename = response.url.split('/')[-1]
        h2s = response.xpath('//h2')
        items = []

        for h2 in h2s:
            country = h2.xpath('span[@class="mw-headline"]/text()')\
                      .extract()
            if country:
                winners = h2.xpath('following-sibling::ol[1]')
                for w in winners.xpath('li'):
                    wdata = {}
                    wdata['link'] = BASE_URL + w.xpath('a/@href')
```

```
                            .extract()[0]
                # Process the winner's bio-page with get_mini_bio
                request = scrapy.Request(wdata['link'],
                              callback=self.get_mini_bio)
                request.meta['item'] = NWinnerItem(**wdata)
                yield request
```

Our `get_mini_bio` method will add any available photo URLs to the `image_urls` list and add all paragraphs of the biography up to the `<p></p>` stop-point to the item's `mini_bio` field:

```
    ...
    def get_mini_bio(self, response):
        """ Get the winner's bio-text and photo """

        BASE_URL_ESCAPED = 'http:\/\/en.wikipedia.org'
        item = response.meta['item']
        item['image_urls'] = []
        img_src = response.xpath(\
            '//table[contains(@class,"infobox")]//img/@src') ❶
        if img_src:
            item['image_urls'] = ['http:' + img_src[0].extract()]
        mini_bio = ''
        paras = response.xpath(
            '//*[@id="mw-content-text"]/p[text() or
            normalize-space(.)=""]').extract() ❷

        for p in paras:
            if p == '<p></p>': # the bio-intros stop-point ❸
                break
            mini_bio += p

        # correct for wiki-links
        mini_bio = mini_bio.replace('href="/wiki', 'href="'
                       + BASE_URL + '/wiki') ❹
        mini_bio = mini_bio.replace('href="#', item['link'] + '#')
        item['mini_bio'] = mini_bio
        yield item
```

❶ Targets the first (and only) image in the table of class `infobox` and gets its source (src) attribute, e.g. `<img src=`*`//upload.wikimedia.org/…/Max_Perutz.jpg…`*

❷ This xpath gets all the paragraphs in the `<div>` with id `mw-content-text`. If the paragraph are empty (`text()` == False) then the `normalize-space(.)` command is used to force the contents of the paragraph (. represents the p-node in question) to an empty string. This is to make sure any empty paragraph

matches the stop-point marking the end of the intro-section of the biography.

❸ Iterates through the available paragraphs, breaking on the empty paragraph stop-point.

❹ Replaces wikipedias internal hrefs (e.g. /wiki/…) with the full addresses our visualisation will need.

With our bio-scraping spider defined, we need to create its complementary pipeline, which will take the image-URLs scraped and convert them into saved images. We'll use Scrapy's images-pipeline for this job.

The `ImagesPipeline` shown in Example 6-6 has two main methods, `get_media_requests`, which generates the requests for the image-URLs, and `item_completed`, called after the requests have been consumed.

Example 6-6. Scraping images with the image-pipeline

```
import scrapy
from scrapy.contrib.pipeline.images import ImagesPipeline
from scrapy.exceptions import DropItem

class NobelImagesPipeline(ImagesPipeline):

    def get_media_requests(self, item, info): ❶

        for image_url in item['image_urls']:
            yield scrapy.Request(image_url)

    def item_completed(self, results, item, info): ❷

        image_paths = [x['path'] for ok, x in results if ok] ❸
        if image_paths:
            item['bio_image'] = image_paths[0]

        return item
```

❶ This takes any image-URLs scraped by our *nwinners_minibio* spider and generates an HTTP request for their content.

❷ After the image-URL requests have been made, the results are delivered to the `item_completed` method.

❸ This Python list-comprehension filters the list of result tuples (of form [(True, Image), (False, Image) …]) for those that were successful and stores their file-paths relative to directory specified by the `IMAGES_STORE` variable in `settings.py`.

Now that we have spider and pipeline defined, we just need to add the pipeline to our `settings.py` module and set the `IMAGES_STORE` variable to the directory we want to save the images in:

```
# nobel_winners/nobel_winners/settings.py

...
ITEM_PIPELINES = {'nobel_winners.pipelines.NobelImagesPipeline':1}
IMAGES_STORE = 'images'
```

Let's run our new spider, from the `nobel_winners` root directory of our project and check its output:

```
$ scrapy crawl nwinners_minibio -o minibios.json
...
2015-... DEBUG: Scraped from <200 http:.../Albert_Claude>
{'image_urls': [],
  'link': u'http://en.wikipedia.org/wiki/Albert_Claude',
  'mini_bio': u'<p><b>Albert Claude</b> (24 August 1899 \u2013...
    <a href="http://en.wikipedia.org/wiki/Belgium"...>Belgian</a>
    <a href="http://en.wikipedia.org/wiki/Medical_doctor"...>
2015-... DEBUG: Scraped from <200 http:.../Brian_P._Schmidt>
{ 'bio_image': 'full/a5f763b828006e704cb291411b8b643bfb91886c.jpg',
  'image_urls': [u'http://upload.wiki...220px-Brian_Schmidt.jpg'],
  'link': u'http://en.wikipedia.org/wiki/Brian_P._Schmidt',
  'mini_bio': u'<p><b>Brian Paul Schmidt</b>...
...
```

We can see that scraping Albert Claude's biography page failed to turn up an image (a quick trip to Wikipedia confirms it's missing) but Brian Schmidt's page came up trumps. The image was stored in `image_urls` and successfully processed, giving the JPG file stored in the images directory we specified with `IMAGE_STORE` with a relative path (`full/a5f763b828006e704cb291411b8b643bfb91886c.jpg`). The file-name is, conveniently enough, a SHA1 hash of the image's URL, which allows the image-pipeline to check for existing images, allowing it to prevent redundant requests etc.

A quick listing of our images directory shows a nice array of Wikipedia Nobel-winner images, ready to be used in our web-visualisation:

```
$ (nobel_winners) tree images
images
└── full
    ├── 0512ae11141584da1262661992a1b05dfb20dd52.jpg
    ├── 092a92689118c16b15b1613751af422439df2850.jpg
    ├── 0b6a8ca56e6ff115b7d30087df9c21da09684db1.jpg
    ├── 1197aa95299a1fec983b3dbdeaeb97a1f7e545c9.jpg
    ├── 1f6fb8e9e2241733da47328291b25bd1a78fa588.jpg
    ├── 272cf1b089c7a28ea0109ad8655bc3ef1c03fb52.jpg
    ├── 28dcc7978d9d5710f0c29d6dfcf09caa7e13a1d0.jpg
    ...
```

As we'll see in ??? we will be placing these in the `static` folder of our web-app, ready to be accessed using the winner's `bio_image` field.

With our images and biography text to hand we've successfully scraped all the targets we set ourselves at the beginning of the chapter (see Example 6-1 and Figure 6-1). Time for a quick summary before moving on to clean this inevitably dirty data with help from Pandas.

Summary

In this chapter we produced two Scrapy spiders which managed to grab the simple statistical data-set of our Nobel winners plus some biographical text and, where available, a photograph, to add some color to the stats. Scrapy is a powerful library which takes care of everything you could need in a full-fledged scraper. Although the work-flow requires a little more effort to implement than some hacking with `BeautifulSoup`, Scrapy has far more power available and for any ambitious scraping really is a no-brainer. All Scrapy spiders follow a standard recipe, which you should now know, and after a while rolling one out for standard scraping tasks is a breeze.

I hope this chapter has conveyed the rather hacky, iterative nature of scraping and some of the quiet satisfaction that can be had from producing relatively clean data from the unpromising mound of stuff so often found on the web. The fact is that now and for the foreseeable future the large majority of interesting data, the fuel to the art+science of data-visualisation, is trapped in a form unusable for the web-based visualisations that are the focus of this book. Scraping is, in this sense, an emancipating endeavour.

The data we scraped, much of it human-edited, will certainly have some errors, from badly formatted dates to categorical anomalies to

missing fields etc.. Making that data presentable is the focus of the next, Pandas-based, chapters. But first we need a little introduction to Pandas and its building block Numpy.